

Delay Insensitive Circuits

*Structures, Semantics,
and Strategies*

Dennis Furey





Copyright © 2019 – 2023 Dennis Furey.

This work is licensed under a [Creative Commons “Attribution-NonCommercial-ShareAlike 4.0 International”](https://creativecommons.org/licenses/by-nc-sa/4.0/) license.



See <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en> for more information. Commercial licenses are available on request.

ISBN: 978-1-9161681-0-7

Plumstead Publishing House, London

... dedicated to Mark Josephs, who patiently mentored a recalcitrant student of this subject

CONTENTS

Contents	4
I Background and Motivation	15
1 Why to Study Delay Insensitive Circuits	17
1.1 Audience	18
1.2 Motivation	19
1.2.1 Technological neutrality	19
1.2.2 Configurable devices	20
1.2.3 Concurrency theory	21
1.3 Random tips on reading this book	22
2 Why Delay Insensitive Design is Challenging	25
2.1 How not to do it with logic gates	25
2.1.1 Towards a reusable implementation	26
2.1.2 A concept of signaling protocols	27
2.2 How not to do it with DI primitives	27
2.2.1 Better building blocks	28
2.2.2 Implications of the current solution	29
2.2.3 Ways forward from the current solution	30
2.3 How not to compromise	32
2.3.1 A two-output primitive	32
2.3.2 DI versus QDI	33
2.3.3 Yet another majority gate	33
2.3.4 Back to the drawing board	35
2.4 Judgment day	37
2.4.1 What a sequencer does	37
2.4.2 How a sequencer enables a majority gate	38
2.4.3 Implications of this solution	38
3 The Lay of the Land	41
3.1 Overview	42
3.2 The process model	42
3.2.1 Process concepts	44
3.2.2 Generality	44

3.2.3	Environments	44
3.3	Block diagrams	44
3.3.1	Notation	45
3.3.2	Methodology	45
3.3.3	Flattening	46
3.4	Towards a process semantics	47
3.4.1	Trace structural composition	47
3.4.2	Deficiencies of a naive trace structural composition	49
3.5	Petri nets	49
3.5.1	Notation and conventions of Petri nets	49
3.5.2	Expressiveness of Petri nets	51
3.5.3	Compositionality of Petri nets	52
3.5.4	Limitations of Petri nets	54
3.6	Procedural description	55
3.6.1	Combinator examples	55
3.6.2	Repetition	58
3.6.3	Conditional execution	60
3.6.4	Adaptation to an environment	61
4	Success	67
4.1	Reachability graphs	67
4.1.1	Example of a reachability graph	68
4.1.2	Reachability graph algorithms	68
4.2	The transducer model	71
4.2.1	Operation	71
4.2.2	Limitations	73
4.2.3	Utility	74
4.3	Traces revisited	75
4.3.1	Progress obligations	75
4.3.2	Quiescent traces	75
4.3.3	Refinement	76
4.3.4	Trace analysis	76
4.4	From transducers to trace recognizers	77
4.4.1	A preliminary subgraph	78
4.4.2	The complete graph	79
4.4.3	Edge cases	79
4.4.4	Other trace recognizers	80
4.5	Interim remarks	83
II	Formal Models	85
5	Petri Net Plumbing	87
5.1	Mathematical conventions	88
5.1.1	Mapping	88
5.1.2	Domains and ranges	88
5.1.3	Cases	89

5.1.4	Ordinals	89
5.2	From Petri nets to processes	90
5.2.1	A concrete model	90
5.2.2	Presets and postsets	91
5.2.3	Hacking the universe	92
5.2.4	Open Petri nets	92
5.2.5	Process models	93
5.3	Editing operations	94
5.3.1	Rewriting	94
5.3.2	Sums	97
5.3.3	Differences	99
5.3.4	Completion	99
5.4	Process combinators	102
5.4.1	Communication	102
5.4.2	Parallel composition	102
5.4.3	Environmental restriction	103
5.4.4	Sequential composition	103
5.4.5	Choice	105
5.4.6	Recursion	111
6	Reachability Graph Wrangling	121
6.1	Math usage	121
6.1.1	Graphs	122
6.1.2	Partitions	127
6.1.3	Ordinals	127
6.2	Initial reachability graph	128
6.2.1	Overview	128
6.2.2	Derivation	129
6.3	Divergence propagation	130
6.3.1	Divergent vertices	132
6.3.2	Disabled inputs	132
6.3.3	Numbered vertices	134
6.4	Anonymous edge reduction	135
6.4.1	Overview	137
6.4.2	Derivation	138
6.5	Redundant path elimination	141
6.5.1	Overview	143
6.5.2	Derivation	144
6.6	Partition fusion	149
6.6.1	Overview	149
6.6.2	Derivation	153
7	Transducer Tuning	157
7.1	Finite automata	157
7.1.1	Sequences	158
7.1.2	Bracket notation	159
7.1.3	State graphs	159

7.1.4	Deterministic finite automata	160
7.1.5	Non-deterministic finite automata	161
7.2	The transducer	161
7.2.1	Overview	162
7.2.2	Derivation	166
7.3	Serial transducers	169
7.3.1	Overview	169
7.3.2	Derivation	172
7.4	Trace recognizers	175
7.4.1	Non-deterministic relational trace recognizer	175
7.4.2	Deterministic relational trace recognizer	177
7.4.3	Behavioral equivalence	179
7.4.4	Alternative extensional descriptions	179
7.5	A canonical form for Petri nets	181
7.5.1	Overview	181
7.5.2	Preparation	186
7.5.3	Specification	191
7.6	Process combinators revisited	193
8	Block Building	195
8.1	On lists	196
8.1.1	Creating a list	196
8.1.2	Deleting from a list	197
8.1.3	Folding over a list	198
8.1.4	Mapping over a list	198
8.1.5	Inverse of a list	198
8.1.6	Flattening a list	200
8.1.7	Transposing a list	200
8.2	Primitive blocks	200
8.3	Hierarchical blocks	201
8.3.1	Block combinators	203
8.3.2	Block algebra	204
8.4	Netlists	205
8.4.1	Conventions about schematics	205
8.4.2	Specifying a schematic by a netlist	206
8.5	From hierarchical blocks to netlists	207
8.5.1	Primitive blocks	208
8.5.2	Non-unit lists	208
8.5.3	Unit lists	209
8.5.4	The transformation	210
8.6	From hierarchical blocks to primitive blocks	210
8.6.1	Non-unit lists	210
8.6.2	Unit lists	211
8.6.3	The transformation	212
8.7	From blocks and netlists to processes	212
8.7.1	Alphabet soup	213

8.7.2	More transformations	215
8.7.3	Generalized refinement	216
8.8	Connection patterns	216
8.8.1	Schematic capture	217
8.8.2	Permutations	221
8.8.3	Generalized terminal rotations	230
8.9	Repetitive structures	233
8.9.1	Arrays	233
8.9.2	Cascades	233
III Module Families		235
9	As Primitive as Can Be	237
9.1	Petri net optimizations	237
9.1.1	Parallel fusion	238
9.1.2	Serial transition fusion	239
9.1.3	Serial place fusion	241
9.1.4	Self-loop place removal	243
9.1.5	Self-loop transition removal	244
9.1.6	Redundant cycle removal	244
9.1.7	Miscellaneous static optimizations	246
9.1.8	The whole mix	247
9.2	Block optimizations	248
9.2.1	Overview	248
9.2.2	Specifications	249
9.3	DI primitives	252
9.3.1	The continuing saga	252
9.3.2	Universality	253
9.3.3	Cardinality and modularity	253
9.3.4	Specifications	254
9.3.5	Implications	260
9.4	Generalized DI primitives	265
9.4.1	Three-terminal primitive generalizations	265
9.4.2	Arbiter generalizations	268
10	Decisions, Decisions	271
10.1	Ordered trees	272
10.1.1	Definition	273
10.1.2	Terminology	273
10.1.3	Computation	273
10.1.4	Notation	274
10.2	Cascading planar decision waits	275
10.2.1	Lateral	275
10.2.2	Bilateral	277
10.2.3	General	281
10.3	Quadrangular decision waits	285

10.3.1 Basic	287
10.3.2 Vertical	291
10.3.3 General	295
10.3.4 A revised planar decision wait generating function	296
10.4 Multidimensional decision waits	298
10.4.1 Dendriform	299
10.4.2 Crossbar	301
10.5 Decision wait transformations	303
10.5.1 Permuting along the axes	304
10.5.2 Permuting the axes	305
10.5.3 Permuting and rotating	306
10.6 Optimized decision waits	306
10.6.1 Global decompositions	307
10.6.2 Quadrangular	308
10.6.3 Dendriform	309
10.6.4 Crossbar	310
10.6.5 General	311
11 Thin on the Ground	315
11.1 Notation	316
11.1.1 Ordinals	316
11.1.2 Transposing	317
11.1.3 Flattening	318
11.1.4 Coordinates	318
11.2 Sparse decision wait transformations	319
11.2.1 Coordinate transformations	319
11.2.2 Permuting along the axes	320
11.2.3 Permuting the axes	321
11.3 Fallback position	322
11.3.1 Degenerate	323
11.3.2 Separable	324
11.4 Planar sparse decision waits	327
11.4.1 Spanning	327
11.4.2 Enmeshed	333
11.5 Multidimensional sparse decision waits	342
11.5.1 Dendriform	343
11.5.2 Crossbar	345
11.6 Optimization	352
11.6.1 Sparse global decompositions	353
11.6.2 General combining form	354
11.6.3 Decomposition strategies	355
11.7 Verification	357
11.7.1 Alphabet ordering	357
11.7.2 Input symbol assignment	357
11.7.3 Output symbol assignment	358
11.7.4 Process specification	358

11.7.5	Correctness	359
12	All About Arbiters	361
12.1	Notation	362
12.1.1	Scalar multiplication	362
12.1.2	Permutations	362
12.1.3	Zippered function application	363
12.1.4	Probability theory	363
12.2	Arbiter decompositions	364
12.2.1	Mesh	364
12.2.2	Dendriform	374
12.2.3	Token ring	381
12.2.4	General	388
12.3	Transfer functions	390
12.3.1	Probability vectors and distributions	391
12.3.2	Incremental transfer function	393
12.3.3	Incremental token distribution	398
12.3.4	Cumulative transfer function	399
12.4	Access patterns	401
12.4.1	Spatial locality	402
12.4.2	Temporal locality	402
12.5	Metrics	405
12.5.1	Expectation	405
12.5.2	Optimization	406
13	Putting the Word Out	411
13.1	Pep talk	411
13.1.1	A two-wire protocol	411
13.1.2	1-hot codes	412
13.1.3	Dual rail codes	412
13.1.4	Constant weight codes	413
13.1.5	General delay insensitive codes	414
13.1.6	Terminology	415
13.2	Encoders	415
13.2.1	Basic	417
13.2.2	Front optimized	418
13.2.3	Back optimized	419
13.3	Decoders	420
13.3.1	Basic	421
13.3.2	Joinable	429
13.3.3	Factorable	430
13.3.4	Partitionable	434
13.3.5	General	436
13.4	Completion detectors	437
13.4.1	Sequencers	437
13.4.2	Majority gates	439
13.4.3	Recurrence	440

13.5 Transcoders	440
13.5.1 Basic	442
13.5.2 Partitionable	444
13.5.3 General	446
14 Working on the Railroad	449
14.1 Arithmetic units	450
14.1.1 Adders	450
14.1.2 Subtracters	458
14.1.3 Buffers	462
14.2 Dual rail to Sperner code conversion	467
14.2.1 Transcoding algorithm	467
14.2.2 Circuit derivation	469
14.3 Sperner to dual rail conversion	473
14.3.1 Preparation	474
14.3.2 Derivation	475
14.4 Parallelism	480
14.4.1 Dual rail toggles	480
14.4.2 Channel demultiplexers	481
14.4.3 Channel multiplexers	482
14.4.4 Micropipeline controllers	484
14.4.5 A parallel transcoder	486
IV Synthesis	489
15 State Based Synthesis	491
15.1 Overview	492
15.1.1 The uncomplicated case	492
15.1.2 Complications	493
15.1.3 Non-quiescent processes	494
15.1.4 Non-deterministically concurrent processes	495
15.2 Transducer types	499
15.2.1 Anti-refined transducers	499
15.2.2 Feedback anti-refined transducers	500
15.3 Basic synthesis	505
15.3.1 Decomposition	507
15.3.2 Building blocks	509
15.3.3 Combining form	516
15.3.4 Loose ends	516
15.4 Input reduction	519
15.4.1 Decomposition	520
15.4.2 Combining form	525
15.5 State reduction	526
15.5.1 Decomposition	527
15.5.2 Combining form	527
15.6 Separation	528

15.6.1	Decomposition	528
15.6.2	Combining form	530
16	Direct Mapping Synthesis	533
16.1	Overview	533
16.2	Mutual recurrences	535
16.2.1	<i>Ad hoc</i> solution	536
16.2.2	Solution by lists of functions	537
16.2.3	Solution by dependence graphs	538
16.3	Refined canonical forms	540
16.4	Decomposition	542
16.5	Interacting state based synthetic communities	543
16.5.1	Places	544
16.5.2	State based transition arrays	545
16.5.3	Communities	547
16.5.4	Combining form	548
16.6	Interacting direct mapped synthetic communities	551
16.6.1	Overview	553
16.6.2	Transitions	554
16.6.3	Lockable transitions	559
16.6.4	Monitors	562
16.6.5	Direct mapped transition arrays	567
16.6.6	Communities	569
16.7	State implosion	569
16.7.1	A naive solution	569
16.7.2	A better solution	570
16.7.3	Concluding remarks	571
V	Appendices	573
A	Supplementary Remarks on Quasi-Delay Insensitivity	575
A.1	CMOS inverters	575
A.2	Unexposed delays	576
A.3	Conclusions	577
B	Complete Partial Orderings and Fixed Points	579
B.1	Theoretical primer	579
B.1.1	Standard fixed point construction	580
B.1.2	Continuity	580
B.1.3	Ordering of functions	581
B.2	Relevance to DI processes	582
B.2.1	CPO Structure	582
B.2.2	Least upper bounds	583
B.2.3	Continuity of process combinators	584
B.3	Further work	585

C	Decision Wait Metrics	587
C.1	Component count	588
C.1.1	Multidimensional	588
C.1.2	Quadrangular	588
C.1.3	Cascading	589
C.2	Critical path length	590
C.2.1	Cascading	592
C.2.2	Quadrangular	594
C.2.3	Dendriform	597
C.2.4	Crossbar	599
C.2.5	General	601
D	Latency Arithmetic	603
D.1	Latencies as a vector space	603
D.2	Comparison of latency vectors	604
D.2.1	Manhattan distances	604
D.2.2	Expected separations	605
D.2.3	Expected wire delays	605
D.3	Parallel combination of latency vectors	606
E	Arbiter Metrics	609
E.1	Contention	609
E.2	Critical path length	612
E.2.1	Tree	612
E.2.2	Mesh	613
E.2.3	Token ring	615
E.2.4	General	618
F	Dual Rail Buffer Cell Theory of Operation	621
	Bibliography	631

Part I

Background and Motivation

We are called to be architects of the future, not its victims.

R. Buckminster Fuller

CHAPTER



WHY TO STUDY DELAY INSENSITIVE CIRCUITS

The quest for performant, robust electronics for data and signal processing applications has often depended on the help of a small close-knit faction of specialists in asynchronous circuit design working on the margins of the broader engineering community [40]. On those rare but dreaded occasions when its technical debt comes due, the deeply held assumption of discrete global time in synchronous design (as opposed to *asynchronous* design) harshly reaffirms the need for their esoteric skills. Sometimes incremental progress is achievable by a combination of synchronous and asynchronous circuitry carefully organized as far as possible to insulate the majority of engineers from unwelcome contingencies. For example, when following the ***Globally Asynchronous Locally Synchronous*** (GALS) methodology [67, 282, 295], the mainly synchronous designer proceeds more warily but otherwise the same as usual for circuits up to a certain size and complexity, and defers reluctantly to asynchronous interface techniques only at the point where the liabilities of synchronous design become impossible to ignore (due to heat dissipation, clock skew, power consumption, metastability, electromagnetic interference, or the prohibitive cost and performance penalties of a global clock distribution network).¹



The ingenuity of the GALS methodology as a response to a given regime of cultural and business constraints can only be admired, but it invites speculation about the greater things that could be achieved if the asynchronous designer were allowed a freer hand. A notable effort in this direction was the investigation of so called ***Delay Insensitive*** (DI) circuits, pioneered mainly during the 1990s. A style of circuits designed from the ground up for asynchronous operation and scalable to any size, they were seen to lend themselves to a tightly cohesive theoretical framework enabling a full complement of rigorously well founded synthesis and verification tools. Among the most

¹See [68, 275] for an introductory overview of asynchronous design, [50, 140] for an anecdotal history from the trenches, and [281] for more recent industrial applications.

promising implications were fewer bugs, less tedium for the designer, lower power consumption, and mechanically checkable semantics-preserving optimization.

1.1 Audience

The aim of this book is to present a curated study of one possible route to the topic of delay insensitivity. While the treatment is meant to be accessible to non-specialists, the level of detail is pitched to enable a sufficiently motivated reader to replicate any of the techniques discussed. Whether a spectator or participant, the ideal reader is envisioned as an *inpert*: the opposite of an expert, self-directed with an eye on the future, not unduly constrained by preconceptions, nor lacking in skepticism, but willing to forgive an occasional departure from convention in the service of a worthy cause. Prior knowledge of digital circuit design is not required and could well be a hindrance, but a basic college undergraduate-level or auto-didactic acquaintance with discrete math is assumed (*e.g.*, functions, sets, relations, graphs, state machines *etc.* [159, 163], and maybe a splash of probability theory for just one chapter). Readers wishing to dig more deeply should find enough material in the bibliography to keep them busy. Where possible, freely available books, dissertations, technical reports, and draft articles are cited in preference to their paywalled equivalents.



While all prospective readers are more than welcome, it is only fair to temper the expectations of some. Working engineers who have come to grief over timing issues at a late stage of an existing project probably would be better served either by the GALS approach cited above or by [267], the best available reference on *Speed Independent* (SI) design (*i.e.*, not quite DI but close), especially for its strong practical emphasis. See [20, 21, 134, 187, 264, 308] for advice on automated tool support. The present text is geared more toward full custom designs, most likely starting from a clean slate.

Nor should job seekers expect their knowledge of this subject to be much of a talking point. In past years at the IEEE “Async” conferences, a team leader from a large well known company beginning with the letter “T” claimed to recruit asynchronous designers under the savvy assumption that they are more capable than most, even if he had no specific need for their asynchronous design skills [192], but learning this subject as a job hunting strategy will remain a long shot until such managerial shrewdness becomes commonplace. It would be more effective to familiarize oneself with industry standard technologies at least to the level of [38] than to rely on dazzling an interviewer with arcana.

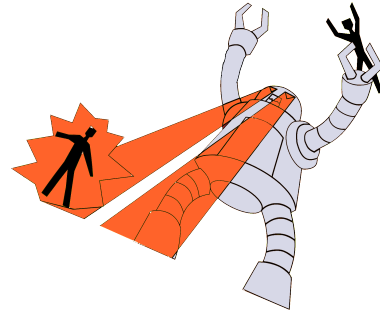
Rather, this book calls for being read in the spirit in which it was written, as an idle diversion for a bored student, engineer, entrepreneur, academic researcher, or maybe even a forward thinking manager given to wondering whether our current working practices are really the best we can do. It may be suitable reading for the summer vacation, the daily commute, the long haul flight, the extended convalescence, or any other off-duty time, especially Part I, which can serve as a self-contained bluffer’s guide to the subject by virtue of soft pedaling the math. Subsequent sections may inspire more technically oriented readers who enjoy working through intricate programming problems in the classic traditions of SICP [272] and TAOCP [143], especially those with interests in compilers, design automation, formal methods, or combinatorial optimization.

1.2 Motivation

The delay insensitive style presented in this book is not the only way of designing circuits, nor even the only way of designing asynchronous circuits. Alternative design philosophies differ in their assumptions about timing [104]. Before exploring the details, it is reasonable for a reader to ask whether DI circuits are worth studying at all, especially with their dearth of recent research activity. Several possible motivations are explained in the remainder of this section.

1.2.1 Technological neutrality

Despite the wave of academic research publications extolling the benefits and “increasing interest” in asynchronous design during the 1990s, there could hardly have been a less opportune time for an upstart to challenge the established practice of digital system development. The steady advance in performance of CMOS technology following Moore’s law [197], with synchronous methods and tools, fueled a juggernaut that posed an insurmountable barrier to entry for almost anything asynchronous, with delay insensitive design possibly the hardest hit. Whatever their supposed merits, delay insensitive circuits were deemed uncompetitive by the market and fell by the wayside as a research topic. However, with the initial trends in speed and device densities having abated somewhat since then, there is a renewed appetite among some technologists to contemplate possible successors to CMOS (e.g. [33, 76, 92, 216, 229, 248, 251]), along with the recognition that familiar assumptions might not always apply to novel contexts.



Modularity

A truism in college engineering curricula holds that a whole computer can be built in principle from nothing but a single type of device called a NAND gate, implying that this device is universal in some sense. In its stark simplicity, the NAND gate alone suffices for any computation whatsoever purely through local interactions with other NAND gates.² An appealing idea, its assertion customarily entails the unstated assumption of a clock, because everyone knows the clock always goes without saying. However, a global clock distribution network, much like scaffolding on an unfinished building, amounts to the biggest, ugliest, costliest, noisiest, least robust non-local interaction of all, and without it the whole flimsy edifice collapses.

What would happen if someone were to take seriously the idea of practical general purpose computation emerging from genuinely local interactions among simple entities, and was prepared even to disregard familiar conventions of circuit design? It turns out that logic gates such as the NAND and other standard modules would become largely irrelevant, but a different class of comparably primitive components would come to light that are more versatile and less tied to any particular implementation technology [81, 136, 223, 294]. This outcome follows naturally from a relaxation of the demand to simulate a timing regime that is fundamentally at odds with physics [83].

The upshot is that while the technological landscape may change, DI circuits travel well. Being quintessentially modular, their semantics is always completely determined by the local interactions of

²A less wrong statement is that a NAND gate suffices to implement any Boolean logic function [137].

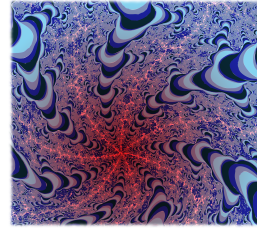
a limited assortment of simple primitive components as intended. The primitive components form a fixed set that can be understood in strictly operational terms by the designer without reference to the underlying implementation technology. Much of the job of porting a design from one manufacturing process to another reduces to porting the primitives.

Migration

At this writing (late 2010s), the successor to CMOS is anyone's guess, and it is prudent to be prepared for whatever may follow. It would be most unfortunate if a future physicist, chemist, biologist, or materials scientist were to overlook some natural phenomenon that could support the implementation of DI primitives (e.g., [46, 205, 224, 238]), and the risk of such setbacks is aggravated by a general lack of awareness of this topic. Specialists in other fields can hardly be expected to second-guess a near universal consensus among electrical engineers that logic gates are the fundamental building blocks of all digital systems, much less to infer the tacit qualification that these devices are practically useless at scale without either some very delicate timing assumptions or a clock. Hence we may sometimes hear of biochemists discovering ingenious ways to implement molecular logic gates [70, 100], but doing so quite possibly in vain by neglecting to discover a molecular clock distribution network to go with them. Alternatively, in the more optimistic scenario of a discoverer of naturally occurring DI primitives recognizing the implications, the intellectual labor to be saved by prevailing on the extant body of knowledge should justify its preservation.

1.2.2 Configurable devices

Often the driving force behind an up and coming technology, from microcomputers during the early years through the current crop of embedded development kits and graphics co-processors, is a devoted following of hobbyists and enthusiasts. **Field Programmable Gate Arrays** (surveyed in [61, 151]) would seem ideally situated to attract similar interest, but have achieved only limited success in that regard at best.³ Although undoubtedly due in part to the marketing focus on commercial developers by the major FPGA vendors, this unfortunate circumstance might be more aptly explained by the hard truth that FPGA programming is a little too much like work to fit most people's idea of a recreational activity. The true hacker's satisfaction derives from seeing the emergent properties of the brainchild take on a life of their own [37]. By contrast, a system that must be dragged, cajoled, prodded, and kicked, only to yield a less interesting return than the sum of the effort expended, lacks this essential allure.



Difficulties with FPGAs

The current state of FPGA technology relegates the bulk of the developer's working life to an endless routine of troubleshooting problems with timing. A correct and complete design with respect to the semantics of the specification language [11, 302] is only the beginning of the real work. It may yet fail in practice due to wire or component latencies that are not modeled in the language. Nor can they be, subject as they are to the vagaries of the subsequent **routing** phase (i.e., the part about connecting it all together). Chaotic phenomena are said to be their own simplest descriptions (hence

³See [91, 303] for a couple of FPGA hobby projects if the links are still alive, and [206] for a contrasting opinion.

impossible to simulate predictively [97, 173]), and support for simulation in FPGA development is tellingly fragmented and proprietary [15, 198]. While pedagogical examples of putting a few gates together may suffice for introductory tutorials and student projects, high performance custom FPGA applications at scale remain the preserve of an elite few who rightly command ample compensation (e.g., [234, 300]), belying the ideal of FPGAs empowering individuals and small enterprises with in-house hardware development capability [13].

Routing is clearly a computationally intensive problem and is not solved as such by delay insensitive design [8, 62, 117, 147, 160, 164, 165, 286]. However, a delay insensitive style of specification would be robust against timing variations introduced by the routing algorithm, and hence would succeed at least where the current state of FPGA technology fails to impart a viable abstraction boundary between specification and implementation. Beyond its philosophical appeal, this simple improvement would have the very practical consequence of putting realistic FPGA development within the grasp of anyone in a position to comprehend a programming language, while incidentally easing the workload of experienced FPGA developers.

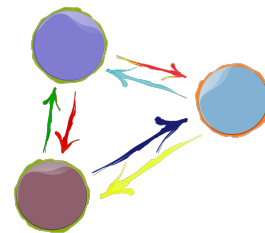
Ways forward for FPGAs

It is not possible to implement a complete set of DI primitives with any combination of the logic elements typically provided by industry standard FPGA boards [31, 105], and hence not currently possible to implement general purpose delay insensitive circuits as FPGAs without building a custom research prototype. Moreover, the dominant FPGA vendors are firmly committed to a course of synchronous architectures with ever more *baroque* embellishments intended to remedy their inherent shortcomings [54, 307]. However, nothing compels a less entrenched stakeholder to follow suit.

- A crowd funded campaign to produce a new delay insensitive FPGA board would attract small enterprises, hobbyists, and advocates of open standards (e.g., [259, 263, 265, 266, 269]).
- Another relevant player might turn out to be a VC funded start-up. While not likely to advance open standards, it would at least create wealth for its founders and investors by catering to an untapped market.
- Furthermore, a moderate level of sporadic interest in asynchronous FPGAs persists among academic research groups, albeit focused mainly on quasi-delay insensitive [90, 177, 228, 283, 304] or other hybrid delay models [119, 167] that do not fully relieve the designer of the need to cope with timing issues.

1.2.3 Concurrency theory

Some of the most important but infamously slippery problems in computer science pertain to concurrent and distributed systems, such as cache coherence [123] and lock free synchronization [107]. Rigorously defined models of concurrent computation have occasionally proved helpful for reasoning about them, including state machines [115], Petri nets [201], and various process description formalisms [36, 106, 113, 156]. Aside from their application to hardware design, delay insensitive circuits occupy a niche in the taxonomy of process networks as a physically realistic model of distributed computation characterized by blocking reads, non-blocking writes, unbuffered channels, and non-deterministic processing elements as network nodes. If the



length of this book is any indication, this simple combination of conditions is also demonstrably fertile. The model deserves coverage in standard computer science curricula for whatever further insights it may bring.

Generalizations of DI circuits

For purposes of hardware design, it is appropriate to limit the processing elements to finitely many states, and the channels to carriers of nullary signals, but these constraints need not apply to other contexts where such a model might be of interest. These might include the design of a process architecture for a real-time embedded control system, or the multi-threaded software implementation of a custom network communication protocol. In these cases, the primitive components would map onto threads of control rather than circuit elements, and the channels would map onto the relevant inter-process communication infrastructure rather than physical wires. Although software applications are not investigated in this book, one might envision verification of such a design against the constraints of delay insensitivity helping to establish its avoidance of dropped packets due to buffer overruns.

An intermediate level of abstraction

Petri nets have been the method of choice to model many problems of this nature with good reason, because desirable and useful system properties such as safety and liveness are usually mechanically verifiable in terms of the Petri net model. Of course, these assurances are reliable only insofar the Petri net model actually corresponds to the system under consideration. A potential strength of delay insensitive circuits is the kind of straightforward mapping between the model and the implementation mentioned above, diminishing the opportunity for human error in identifying one with the other. Moreover, as introduced in [Chapter 3](#) and explored further in Part III, DI circuits can be interpreted as a restricted form of Petri nets, allowing automated semantics-preserving transformation between the two and combining their advantages.

1.3 Random tips on reading this book

There are four main parts to the book and several appendices, with material allocated to the latter based on various criteria. Some of it is not specifically topical for the main text, and some is more speculative. Some may be too detailed to interest the majority of readers, and not all of it adheres to the aforementioned prerequisite of an undergraduate-level discrete math background. Giving the appendices a miss would not in itself dilute the main message of the book, but reading them may provide some ideas for further work in this area.

Some readers, otherwise known as graduate students, reputedly prefer the bibliography as their first point of entry to any publication. To assist this audience, every bibliographical reference lists the pages in the text where it is cited. In electronic versions of the book, these page numbers are clickable hyperlinks. In this way, a reader can quickly gauge its credibility by scanning the bibliography for a familiar name or title and jumping to the relevant section to ascertain whether the reader's interpretation of the cited work accords with the author's (which is normally that it corroborates, amplifies or exemplifies a point being made).



Some readers may wonder whether it is worth charging through all the math just to pick up a few circuit design ideas. Despite appearances, this book is not a math book; it employs mathematical notation only as needed to communicate unambiguously.⁴ Perhaps this claim will meet with skepticism from non-mathematicians. To test it, any reader thus inclined is invited to ignore the math and just look at the circuit diagrams, go off and code them up in VHDL or Verilog [214, 225], preferably run some simulations, and return to the text for a few pointers only in the event of any unforeseen difficulty.⁵ Similarly, software developers may elect to code a few of the algorithms in their preferred style by referring only to the informal narrative and ignoring anything with lambda notation or an equation number in it. If there turns out to be no difficulty whatsoever, then by all means write up an appendix to that effect and submit a pull request.

Each chapter or appendix ends with a short series of questions limited to one page. These questions include verbatim reminders of key points, straightforward tests of understanding, kōans (i.e., deliberately ill-posed questions whose contemplation elicits insight), conversation starters, coffee break sized problems, weekend projects, and open ended projects. Undoubtedly a reader will be more drawn to some than others, but should be extremely cautious about assuming a question is easy without actually answering it. This subject seems to have an unusual affinity for simple questions whose answers demand serious thought, and some of the most innocuous questions posed here are distilled from epic historical debates. Nevertheless, because this chapter is only the first, we can start with some genuinely easy ones with no imminent cause for alarm if any of the answers is not yet clear.

Easy-peasy questions

1. What is the difference between synchronous and asynchronous circuits?
2. How long has asynchronous design been going on? (A lower bound is sufficient.)
3. By what criteria should a circuit designer choose among DI, SI, and GALS?
4. What is a DI primitive?
5. Why should circuit designers care about technology migration?
6. How street-credible is the author's take on FPGAs?
7. In light of the footnote on page 19, are the remarks about universality of NAND gates refuted by the argument that any finite computation can be realized in principle by a giant look-up table? (hint: See [118] or look up the term "*hazard-free*".)



⁴Indeed, DI circuits are virgin territory for anyone interested in automated theorem proving. Synchronous operation is not even a stated axiom in mainstream work on verified hardware, but a fixture of the formalism itself [207].

⁵not to claim this is possible, but presumably a grumpy enough reader will be the judge of that

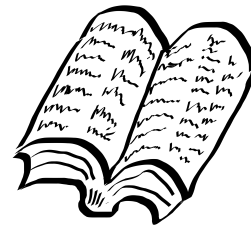
Whenever you find yourself on the side of the majority, it is time to pause and reflect.

Mark Twain

CHAPTER 

WHY DELAY INSENSITIVE DESIGN IS CHALLENGING

A running example in this chapter serves as a gentle introduction to some of the subtleties that distinguish delay insensitive design as a discipline worthy of a book-length exposition. These ideas are refined through a series of unsuccessful but progressively more careful iterations on a deceptively simple delay insensitive design problem. This chapter is written both for readers with no prior knowledge of circuit design, and for those who would like to know why more familiar approaches to circuit design are inadequate to ensure delay insensitivity.



The task is to design what is called a 2-of-3 *majority gate*. It has three input terminals and one output terminal. Whenever it receives a request on any two input terminals, it emits an acknowledgment on the output. Although it is perhaps not the most sophisticated application in itself, it brings a great many important issues into sharp focus, and it provides a context for the graceful *début* of a fair amount of jargon and terminology.

2.1 How not to do it with logic gates

To avoid any misunderstanding, let the input terminals be named a , b , and c , and let the output be named x . Then the requirement can be restated that concurrent inputs to a and b , or to a and c , or to b and c , should cause an output to appear on x . To be even more precise, we can rephrase this statement as a logical equivalence using symbols \wedge for conjunction (*i.e.*, meaning both left and right operands are true) and \vee for disjunction (when either is true).

$$x \Leftrightarrow (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \tag{2.1}$$

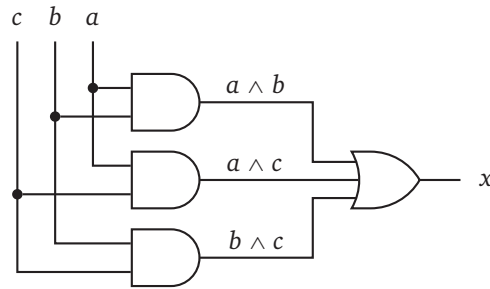


Figure 2.1: logic circuit for a 2-of-3 majority gate corresponding to [Equation 2.1](#)

A normal college engineering textbook style of solution would draw inspiration from this logical relationship to construct the circuit shown in [Figure 2.1](#). Each of the inputs a , b and c is carried by a forked wire to two of the three D-shaped devices known as AND gates, which evaluate the conjunctions of their inputs. The intermediate results $a \wedge b$ etc. are transmitted by them to the remaining device, an OR gate, which computes the disjunction of its three inputs, and transmits the result to the output x .

Although the logic may seem unassailable, the crucial test of this design hinges on the operative word “whenever” in the specification, *i.e.*, on repeatably yielding correct results. To claim success when initial concurrent inputs of a and b are acknowledged by the output x , followed by a subsequent input of c alone that is not, presupposes some mutually agreed protocol between the circuit and its user to distinguish one usage from the next. If another input of b were to follow immediately, the required output behavior would differ depending on whether the latter b were counted in isolation or in conjunction with the preceding c . At a minimum, more consideration is necessary before declaring this problem solved.

2.1.1 Towards a reusable implementation

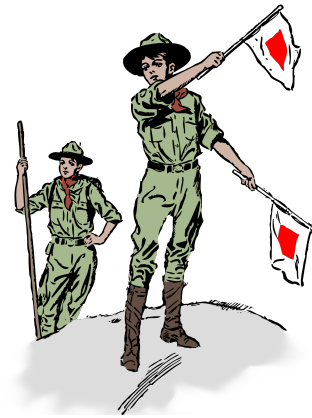
A traditional approach to resolving the current issue would be to revise the specification by imposing time constraints on both the user and the circuit. To be regarded as concurrent, two of a , b , or c must be transmitted by the user either at exactly the same time (arguably a zero-probability event) or within some limited interval of each other. An acknowledgment on x (or lack thereof) must be given by the circuit within yet another specified interval. The implementation could then be deemed correct if and only if it performs according to this revised specification. How these conditions might be enforced in practice is another matter, but if this solution could be made to work, it would be called a **bounded delay** model design. Failing that, the next step in this direction is simply a synchronous design with all of its attendant pitfalls.

A more promising approach eschews fixed time constraints in favor of delimiting the interactions between the user and the circuit by its natural rhythm. After transmitting a pair of inputs, the user is required to wait for the acknowledgment from the circuit before proceeding. However, this solution raises a related issue to the one noted above when one of the same inputs (*e.g.*, b) is used in each of two consecutive interactions. Without properly formulated signal coding conventions, two short consecutive signals on the same input could be indistinguishable from one long one.

2.1.2 A concept of signaling protocols

An explicit statement of something usually taken for granted may help to elucidate this issue. With networks of logic gates such as those used in this circuit, standard practice associates one of two states (*e.g.*, true or false) with each wire at any moment. Each gate is modeled by a function that maps each possible set of current input states to a corresponding output, either true or false, depending on the particular semantics of the gate being modeled (*e.g.*, whether it is an AND gate or an OR gate).

In the intuitive justification given above for the circuit in Figure 2.1, a signal is tacitly identified with a true state, and the absence of a signal with a false state. Two consecutive occurrences of the same signal therefore would correspond to a wire holding a true state unchanged for the duration. While this coding convention might be made to work in a synchronous design, it is insufficient as an asynchronous protocol, which would depend on always being able to detect the arrival of a fresh signal.



4 Φ signaling

One well known way of working around this problem has been to require that all inputs and outputs return to the false state at the end of each interaction. This practice, sometimes known as level signaling or 4 Φ (four phase) signaling, becomes something of an organizing principle in the theory of Null Convention Logic (NCL) [85, 86]. However, this remedy alone would not suffice in the current setting without further revisions to the circuit, because as it stands, dropping either of two true inputs to false could cause the output to drop prematurely before the other input is dropped, thereby misleading the user into thinking the circuit is ready for another interaction.

2 Φ signaling

Another workaround would be to encode a signal as the change from true to false or from false to true, without regard for the wire's logical state at any time between changes and hence without distinguishing between the two possible directions of a change. This signaling convention, sometimes called NRZ (non-return-to-zero), transition signaling, or 2 Φ (two phase) signaling, was advocated influentially in [273]. An advantage of this technique is that less time and energy are wasted on redundant activity, but employing it in this example would require an even more radical redesign of the circuit.

2.2 How not to do it with DI primitives

These difficulties might have been avoided with more forethought. If the most natural statement of the specification is in terms of the signals the circuit sends and receives, it is appropriate to build it from components that can be understood similarly. The AND gates and OR gate used in the first iteration of this design, being defined only as maps from steady state input logic levels to output levels, appear in retrospect to be the wrong tools for the job.

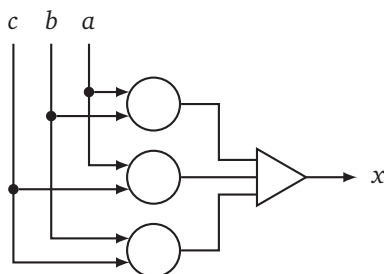


Figure 2.2: improved majority gate using DI primitives

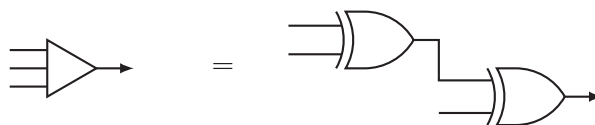


Figure 2.3: implementation of a MERGE by XOR gates

2.2.1 Better building blocks

What we really need instead of an AND gate is a device that detects a confluence of *signals*. That is, it should wait for a signal from each of its inputs, which may arrive at different times, and then send a signal on its output. This behavior requires it to have some form of internal memory or state, which precludes an implementation by any device whose output is fully determined by its current input alone. In place of an OR gate, we need a device that merges two streams of signals. That is, it must wait for a signal on either of its inputs and then send it to the output.

Assuming for the moment the availability of devices having these imprecisely stated properties, we might redesign the circuit as shown in Figure 2.2. The circular schematic symbol standing in for the AND gates in the original design is sometimes called a C gate, a C element, a Muller C element, or even a *rendez-vous* elsewhere, but is designated more briefly and descriptively as a JOIN in this book (following [42, 78, 136] among others). The triangular component replacing the OR gate from the previous design is known as a MERGE.

With this step, it also becomes possible to discuss DI circuits in a technology-independent way. Whether a signal is conveyed by an inversion in logic level, a change in the concentration of chemical solution, or a projectile blown through a tube is immaterial provided the components relay the signal the same way.¹ However, their implementations in conventional technology with 2Φ signaling may merit a momentary digression.

Implementation of a MERGE

The familiar two-input logic element known as an XOR gate (as in “exclusive or”) outputs a true value whenever either but not both of its inputs is true. However, a device with this logic function

¹In the case of a signal mediated by a physical particle, an active particle-duplicating FORK device is also needed.

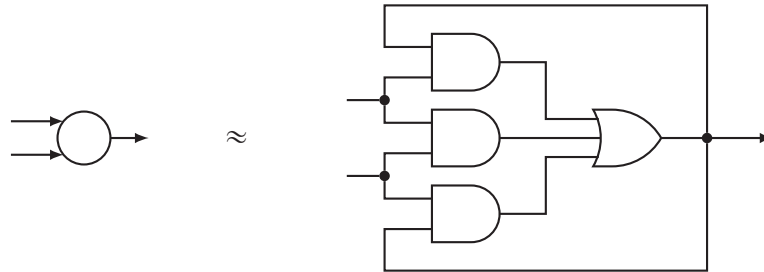


Figure 2.4: approximate implementation of a JOIN by AND gates and an OR gate

can also serve a different purpose in the context of 2Φ signaling. An ordinary two-input XOR gate performs a 2-way merge because a change to either of its inputs in either direction results in a change to its output (not necessarily in the same direction). The merging of three or more channels can be achieved by forming a tree of XOR gates (*i.e.*, by connecting the outputs of some gates to the inputs of others as in Figure 2.3).

Although it can be shown by Boolean algebra or truth tables that a tree of XOR gates does not generally output the appropriate truth value for a multi-way XOR logic *function*, it readily suits our present purpose of merging streams of level *transitions* (barring contention). As 2Φ DI designers, we care only about the logic level changes themselves, not the values they attain. A rigorous justification of the latter claim using only Boolean algebra would be a chore, but a *gestalt* shift to 2Φ thinking makes this proposition easily intuitive.

Implementation of a JOIN

As noted above, no logic gate can implement a JOIN by itself. However, a serviceable implementation subject to certain caveats can be attempted using the circuit shown in Figure 2.4. The intuition is that an initially low (*i.e.*, false) output remains so until both inputs force the central AND gate high (*i.e.*, true), and then the output feeds back through either of the other AND gates to hold itself high until both inputs drop.

There are two possible problems with a gate level implementation of a JOIN primitive. One is a race condition. The circuit could malfunction if the propagation delay along the internal feedback paths significantly exceeds the time taken for the output signal to be received externally. In this case, the JOIN performs no differently than an AND gate for the duration of some unspecified time interval following the first rising output (with more complicated knock on effects possible thereafter), contrary to the intended signaling protocol. Correcting this problem might require timing analysis based on physical parameters beyond the scope of logic design. The other problem is inefficiency. A custom designed implementation is likely both to outperform and to require less hardware than an equivalent network of logic gates, at least in CMOS technology [250].

2.2.2 Implications of the current solution

Leaving aside potential implementation issues with these devices, we return to the question of whether the solution in Figure 2.2 successfully implements a 2-of-3 majority gate. There is good news and bad news.

- The good news is that with the first arrival of signals a and b , the uppermost JOIN sends a signal to the MERGE, which relays it to the output as required. In contrast to the previous iteration of this design in Figure 2.1, a subsequent (2Φ convention) input on a or b does not immediately affect the output from the MERGE until a signal on another input is received, also in accordance with the specification. Similar reasoning applies to any of the three possible pairs of inputs.
- The bad news is that this circuit does not cope with the case of an initial input pair followed by a different input pair. For example, if initial inputs of a and b are acknowledged by the output, and then inputs b and c are applied, the behavior of the circuit is undefined. The problem is due to the b input being forked to both the upper JOIN and the lower one, and therefore being received by the latter both times.
 - If the c precedes the second b , the lower JOIN can output immediately, having already received the first b , and cause the MERGE to output prematurely.
 - If the c lags the second b , there will have been two consecutive signals due to b received by the same input to the lower JOIN, with no signal received by the other input. The behavior of the JOIN itself under these circumstances is undefined (*i.e.*, unpredictable).

2.2.3 Ways forward from the current solution

A skeptical reader might consider blaming the failure of this design on a point of pedantry in its construction. Because a JOIN is just an abstraction, it should be straightforward to resolve the last issue mentioned above by stipulating henceforth that a JOIN may indeed receive two consecutive signals on the same input.² In the case of two consecutive signals on the same input, the definition of the JOIN could be extended to prescribe a reversion to its previous state. Reverting to its previous state is what “really” happens anyway based on the gate level implementation in Figure 2.4, provided we assume that signals are conveyed by changes in logic level and the previous state happens to be where both inputs are false. Such a device is not unknown, and is sometimes called an NCEL [79].

4 Φ solution using an NCEL

Using this newly defined component, a pragmatic engineer can forge ahead to rescue this design by insisting on a four phase protocol. The initial inputs of a and b from the user (for example), followed by an acknowledgment x from the circuit, are to be followed necessarily by another transmission of a and b from the user to cancel the extra a that went to the middle NCEL (previously a JOIN) and the extra b that went to the lower NCEL. Precisely these cancellations, previously ruled out in the case of a JOIN, are now legitimately within the specification of an NCEL. This second pair of inputs also triggers the upper NCEL again, and is consequently acknowledged by another x from the circuit, indicating that the circuit has returned fully to its initial state and is ready for any new pair of inputs.

If doubling the latency and energy dissipation per operation by this latest innovation is of no consequence, perhaps introducing a race condition is harder to discount. The sticking point is that this proposed remedy enforces no particular deadline for any of the “extra” signals mentioned above to be canceled. If the latter phase inputs of a and b propagate more swiftly though the top NCEL

²There are well informed theoretical arguments against this change to the definition [288]. Its inadequacy to solve the present problem suffices for this discussion.

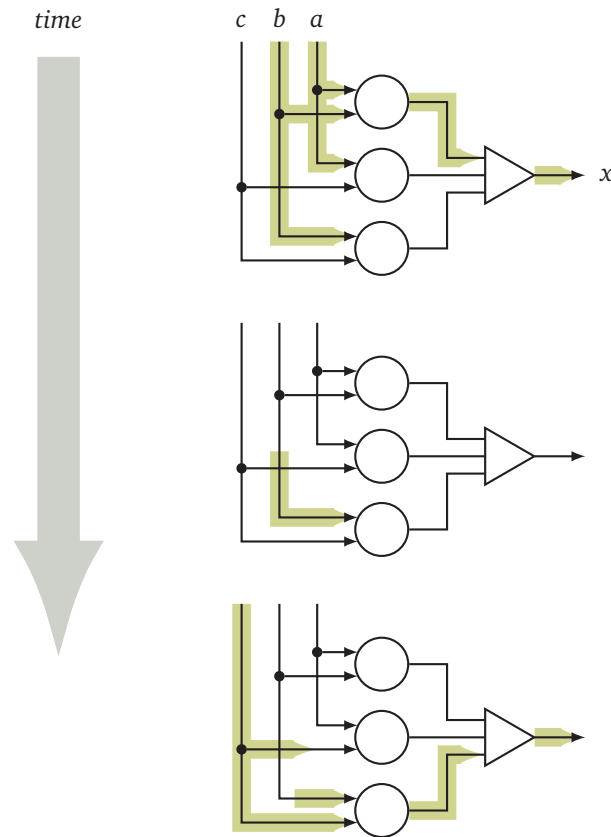


Figure 2.5: First, concurrent inputs of a and b are acknowledged by x (top). Then the resets of a and b are prematurely acknowledged by that of x (center). Finally, an input of c all by itself is wrongly acknowledged (bottom).

and outgoing MERGE than along their respective internal branches, there is still the same possibility of a subsequent input reaching a partially charged NCEL as in the case of a JOIN before.

This syle of reasoning about a circuit normally can be somewhat difficult, but fortunately if the signals are carried by worms leaving a trail of glowing radioactive slime along the wires they traverse until it dissipates, the failure mode described above is easy to visualize. As shown in Figure 2.5, the slime trail along the branch from the input b to the bottom JOIN or NCEL is stickier than the others, and therein lies the trouble.



Speed-independent design

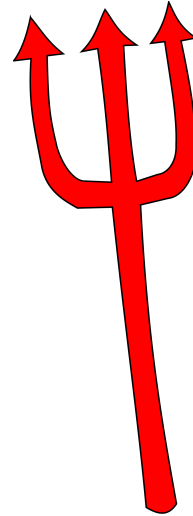
If still not yet deterred, one could continue in this vein along the lines of another well trodden workaround for this sort of problem. The job of an asynchronous designer would be so much easier if only the signals propagating through every fork were accommodating enough to reach both ends

at the same time. This state of affairs with a bit of help would suffice to salvage the current situation, for example. Fortunately, it is not just a matter of wishful thinking but the basis of an actual design philosophy called **Speed Independence**, or SI [148, 149, 218, 267]. Its plausibility relies on the premise that two events occurring within a minimum gate delay of each other can be regarded as simultaneous for practical purposes. Hence, to justify the assumptions of speed-independent design, it is necessary to establish that wire delays are negligible compared to component latencies. That is, the propagation delay associated with the longest wire anywhere in the system must be materially less than the latency of the fastest gate or component anywhere else in the system.

Regardless of the technology, the propagation delay of any physical communication channel can be expected to increase at least in proportion to its length,³ while component latencies race to the bottom with each new generation, pitting these two parameters against each other in any sufficiently complex system [111]. Similarly to the implications of a global clock distribution network in synchronous systems, some limitation on scalability must be anticipated as a fixture of SI design.

Quasi-delay insensitive design

A variation on SI potentially escaping this limitation could provide for some but not necessarily all forks to have this special relationship with the signals they carry, amounting in practice to a requirement for the difference in propagation delays between the prongs to be dominated by that of the shortest feedback path.⁴ Each of these so called **isochronic** forks must be verified individually by recourse to physical measurements, rules of thumb, gut feelings, or other techniques extrinsic to the circuit's logical organization [24]. This *modus operandi* is known as **quasi-delay insensitive** (QDI) design.



2.3 How not to compromise

QDI design was most influentially advocated originally as a pragmatic necessity based on a formal argument characterizing the class of delay insensitive circuits as severely limited, with isochronic forks being the “weakest possible compromise” needed for them to be of any real use [182]. While our present difficulties would appear to support this conclusion, due diligence before abandoning this project requires attending to a condition noted by several authors since then, whereby this argument was arbitrarily restricted to circuits made from forks and single-output devices [44, 68, 104, 210]. Further consideration of this matter follows in Section 2.3.1 and Section 2.3.2, with a revised design attempt after that.

2.3.1 A two-output primitive

An example of something other than a single-output device known at least as early as [79] is the TOGGLE. As shown in Figure 2.6, it has one input terminal and two output terminals, with one of the output terminals distinguished by a dot in the schematic symbol. The first time a TOGGLE receives a signal on the input terminal, it acknowledges the input signal by sending an output signal from

³barring wormholes, time travel, teleportation, temporal anomalies, warp drive, etc. [166, 199, 211]

⁴See [138, 178] for a definition of isochronic forks by a slightly less restrictive but more technical delay assumption.

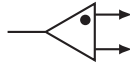


Figure 2.6: schematic symbol of a TOGGLE, an example of a multiple-output device

the dotted terminal. The second time it receives an input signal, it sends an acknowledgment from the other output. The third time, it uses the dotted output again to acknowledge the input, and continues to alternate between the outputs in this way for each subsequent input. This behavior is illustrated in [Figure 2.7](#).

Readers familiar with synchronous design may notice a similarity between the TOGGLE and the conventional T-flip-flop element [38], a state-holding device to be found in any logic designer’s metaphorical tool box. As incidental as it may seem to the deeper questions at hand, lifting the prohibition against the humble TOGGLE is enough to disqualify the often cited but seldom examined case against delay insensitivity mentioned above, because a general analysis involving multiple-output devices is expressly beyond its scope.

2.3.2 DI versus QDI

In deference to the QDI school, one hastens to add that any real implementation of a TOGGLE in standard logic gates or even at the transistor level would certainly depend on isochronic forks in one way or another. Presumably the device must attain internal stability before emitting an acknowledgment signifying its readiness to receive another input. Verification of such a condition is bound to entail some form of physical timing analysis to confirm simultaneity or other temporal relationships among multiple internal events. It could be argued that the use of isochronic forks therefore is not really avoided by postulating multiple-output devices, so the original claim stands.

The best chance of reconciling the philosophies of DI and QDI design might be to regard DI design as a separate paradigm where these concerns, while valid, are less prominent and others more so. By definition, a primitive such as a TOGGLE is not expected to have a DI implementation in terms of other primitives. Moreover, if a circuit is made of primitives concealing isochronic forks, its technology migration is easier than it would be if they had been exposed (*cf.* [Section 1.2.1](#)), especially if isochronic forks are not needed for implementing the primitives in the target technology. If technology migration is of no interest or if isochronic forks turn out to be common to all present and future technologies (a big “if”), there is some merit nevertheless in preventing a complication properly belonging to a lower abstraction layer from insinuating itself into the daily design and verification work flow. For the class of DI circuits *with unrestricted output arity*, the timing analysis required to verify the isochronic forks inherent in the primitives would be done at most once, when the set of primitives is standardized in a reusable component library (as argued eloquently in [294]). See [Appendix A](#) for supplementary remarks on this subject.

2.3.3 Yet another majority gate

To return to our running example, the majority gate design in [Figure 2.8](#) avoids the race conditions plaguing the first two iterations and performs correctly under a broader range of conditions by dispensing with all forks. An input signal on any of a , b , or c propagates through the MERGE to the TOGGLE. Upon receipt of this first input signal, the TOGGLE acknowledges it on the dotted output,

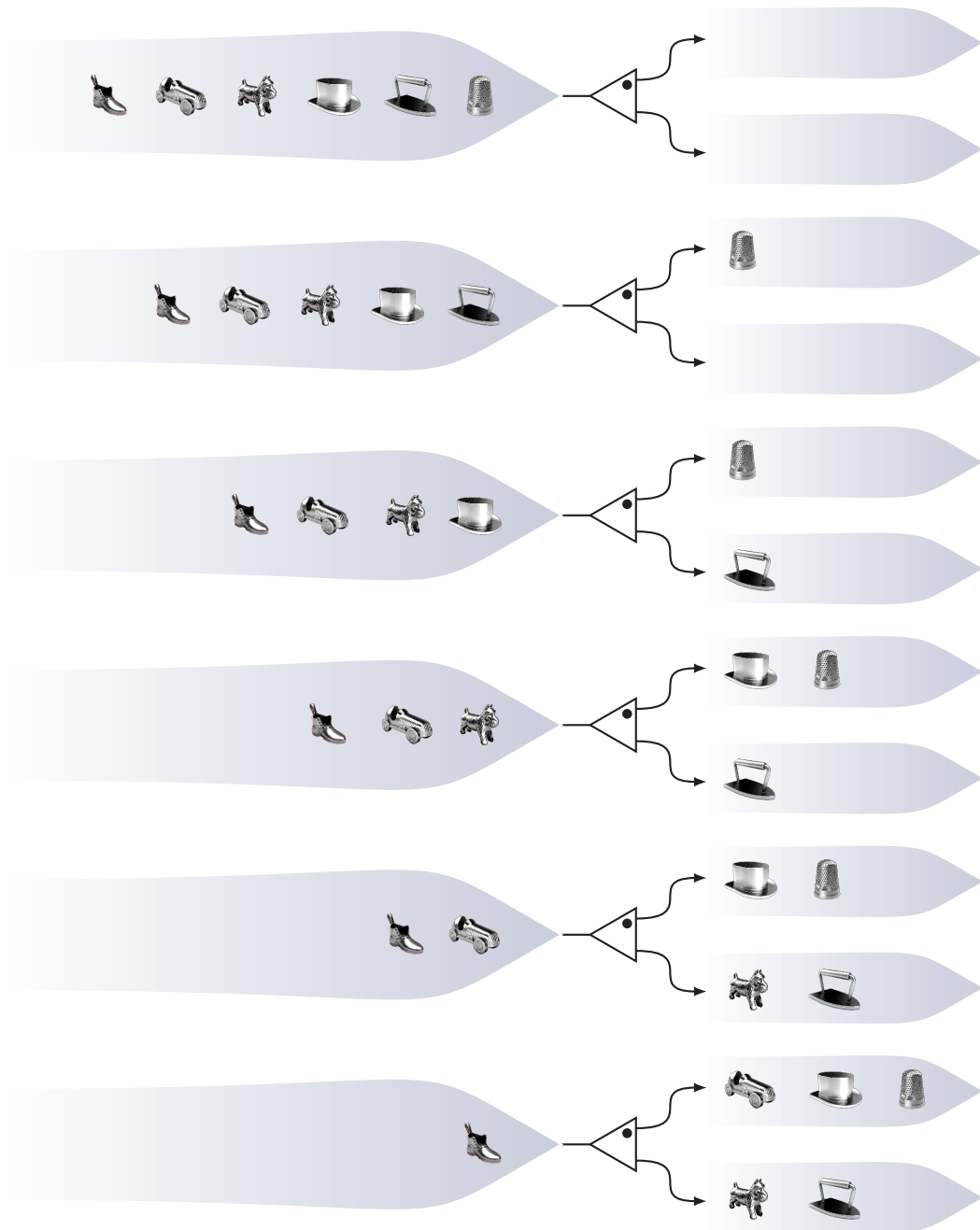


Figure 2.7: six snapshots animating the way signals, visualized as distinct particles, flow from left to right through a TOGGLE, with each successive input signal routed to the alternate output

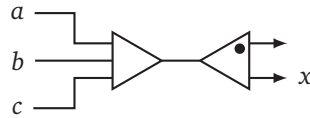


Figure 2.8: third attempt at a majority gate, made from a MERGE and a TOGGLE

which is floated or ignored. A second signal on any of a , b or c propagates again to the TOGGLE, but this time the acknowledgment is sent to the other output, x , because the TOGGLE alternates between its outputs. The next input signal, whatever it may be, leads to an acknowledgment on the floating output, but the one after is acknowledged on x , and so on for each subsequent pair of inputs. The user therefore observes an acknowledgment on x for each pair of inputs.

Unfortunately, this solution does not conform exactly to the original 2-of-3 majority gate specification.

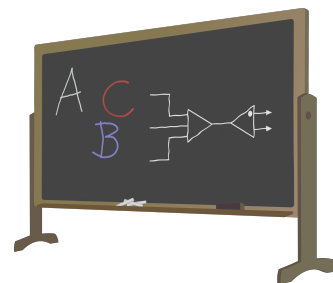
- A minor issue is that two consecutive signals on the same input could also cause an acknowledgment. This behavior is not required by the specification, but on the other hand, it is not explicitly prohibited.⁵ If this extra feature were the only point of discrepancy, the implementation would be called a *refinement* of the specification.
- A more serious issue is that this circuit reacts unpredictably to contention between concurrent inputs (e.g., if inputs on a and b occur at the same time). The behavior of a MERGE is undefined if two input signals are sent to it concurrently.

2.3.4 Back to the drawing board

Several possible ways to improve on this design are considered below, leading to the conclusion that a fresh start with still more care and attention would be best.

Semantic enhancements

Similarly to the previous solution, this one may appear to fail due to a point of semantics. Whereas previously a JOIN was undefined for sequential signals on the same input, in this case a MERGE is undefined for concurrent signals on different inputs. Nor will attempted extensions to the definition be any more fruitful in this case. If a MERGE is implemented by XOR gates as in Figure 2.3, then simultaneous 2Φ inputs can be shown by Boolean algebra to cancel each other, and nearly simultaneous inputs to cause a transient or indistinct output. Furthermore, there is no obviously reasonable way even in a technology-independent sense to extend the definition of a MERGE to cover concurrent inputs.



⁵However, consecutive identical inputs without an intervening output are always *implicitly* prohibited in any delay insensitive specification [288].

Fundamental mode

Less like correcting the defects of this design than banishing them by decree, we may simply insist that only one input at a time be provided to the circuit, and that sufficient time be allotted between inputs for the circuit to reach a stable state internally. If this length of time is not indicated to the user by observable signals from the circuit, it will have to be divined by other means. This timing regime, known as *fundamental mode*,⁶ has been suggested as a way of adapting synchronous logic to the design of a limited class of asynchronous state machines [284]. Among the earliest work in asynchronous design, fundamental mode remains an enduring staple of optional chapters in college engineering textbooks along with stern warnings against anything asynchronous [38].

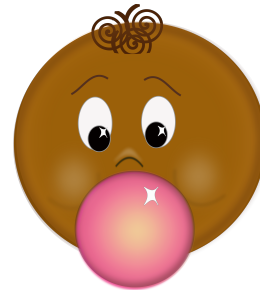
For our purposes, fundamental mode raises the more pointed question of whether it is consistent with the spirit of the original specification. Although a circuit designed according to fundamental mode assumptions could conceivably adhere to a delay insensitive signaling protocol when interacting with its environment, it happens not to be possible for this particular application (related discussion in [42]). By way of a brief hand-waving explanation, the notion of a stable state is not rigorously established in the context of a delay insensitive circuit specification given by the interface between a system and its environment.⁷ To interact delay insensitively with a fundamental mode circuit, the user must be reliably informed as to its “stability” (*i.e.*, its readiness to receive an input signal), effectively requiring every input signal to be acknowledged individually, which is contrary to the majority gate specification.

Burst mode

Newer methodologies known as *burst mode* [210] and *extended burst mode* [311] have removed some of the limitations of fundamental mode on state machine synthesis. In burst mode specifications, a prescribed set of input signals may occur concurrently and be acknowledged as a unit. Burst mode designs have benefited in the past from a relatively well established infrastructure of end-to-end CAD support and technology mapping [124], albeit based on software whose current maintenance status, availability, and license conditions are unclear.

Although the range of behavior encompassed by burst mode specifications is less restricted than that of fundamental mode, it is not completely general. Input signals may not occur concurrently with outputs, non-deterministic choice and arbitration are excluded, and it remains incumbent on the user not to intrude on the circuits thus synthesized at times of instability. Whether a formal mandate or not, this last condition is a *de facto* feature of the model in combination with the existing tool set, arguably due to its reliance on standard logic gates.

Fortunately, a majority gate fits well within the scope of expressible burst mode specifications. Relinquishing an engineering design job to burst mode synthesis tools is an option whenever the fundamental mode environmental condition and associated behavioral limitations are acceptable. However, to fulfill the current brief of a delay insensitive design, another approach is necessary.



⁶The term “fundamental mode” may have been chosen originally for its connotation of a system not driven beyond its minimum resonant frequency. In modern parlance, the opposite of fundamental mode is the regrettable coinage “i/o mode”.

⁷Some alternative theories of delay insensitivity have explicitly invoked a formal concept of state (most notably [43, 172]), but an extensional theory is preferred in this text for a more parsimonious refinement relation among other reasons.

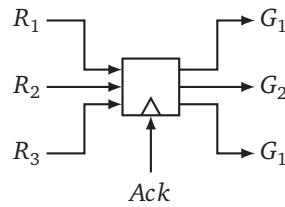


Figure 2.9: a three way sequencer

2.4 Judgment day

Although the design in Figure 2.8 does not meet the specification, it performs correctly under certain restricted conditions (namely the absence of contention) and may be refined into a correct solution. In many DI applications, failures due to contention among concurrent inputs can be prevented by arbitration. To incorporate arbitration into the current design, a device not previously introduced known as a **sequencer** is needed (sometimes also called an arbiter in other sources).⁸ A leisurely explanation is in order here because this concept tends not to be widely appreciated [49, 50].

2.4.1 What a sequencer does

The particular form of sequencer shown in Figure 2.9 is a three way sequencer. Each of the three inputs R_1 through R_3 is associated with one of the three outputs G_1 through G_3 (mnemonic for “request” and “grant”). The input labeled *Ack* on the bottom depicted as a stylized caret is special. The first time the sequencer receives a signal from one of R_1 , R_2 or R_3 , it sends a signal to the corresponding output G_1 , G_2 , or G_3 . After that, it waits for an acknowledgment on the *Ack* input. If that acknowledgment is the next thing to happen, then life is easy and the sequencer resumes its former slumber until obliged to repeat this process by another request.

A sequencer always reacts eventually to every request by sending a signal to the corresponding grant, and then always waits for an acknowledgment, but things get complicated if new requests arrive before the sequencer is finished with the current grant or acknowledgment cycle.

- One possible complication would be for a second request to arrive after the sequencer issues the first grant but before it receives an acknowledgment. In this case, the sequencer does not yet grant the new request, but continues waiting for the forthcoming acknowledgment. The sequencer never sends a grant while an acknowledgment is pending. When the acknowledgment finally arrives, the sequencer then issues the grant it has been withholding.
- A similar situation occurs when the sequencer receives two requests within such a short interval that it has not even granted the first one before the second one arrives. In this case, the sequencer grants the first request and then proceeds as above.
- Yet another possibility, however unlikely, is that two requests arrive at exactly the same time. In this case, the sequencer makes an arbitrary, non-deterministic choice about which request

⁸There is no unanimous consensus on terminology. The sequencer is not treated as a primitive in this book and is regarded as distinct from an arbiter, which is a primitive. A sequencer is constructed from simpler primitives including an arbiter.

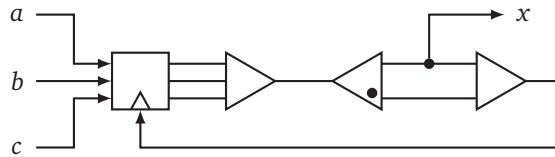


Figure 2.10: a correct 2-of-3 majority gate using a sequencer

to grant first, and acts as if the other request had arrived second. This capability is the most important feature of a sequencer.

- More pathological cases are not ruled out. If three requests arrive initially at the same time, the sequencer grants only one request at first. After the sequencer receives an acknowledgment to the first grant, it chooses one at random between the two remaining requests and grants it. After the sequencer receives the acknowledgment to this latest grant, it finally grants sole remaining request.

In summary, the sequencer is an asynchronous designer's secret weapon. It takes signals almost any way the environment can throw them at it and lets them through one at a time in an orderly queue. The only way to derail a sequencer is to send the same requesting signal to it repeatedly without waiting for the corresponding grant output inbetween, but a similar rule applies to any delay insensitive device [288].

2.4.2 How a sequencer enables a majority gate

The sequencer in Figure 2.10 solves the problem with contention from the previous design. To understand how, it may be helpful to imagine signals flowing through the circuit like physical particles or to perform a worm-level simulation in the style previously noted (page 31). Whether any two input signals a , b or c arrive separately or together, one gets through the sequencer, while the other, if any, is detained. The one that gets through then proceeds through the first MERGE, then through the TOGGLE, out by the dotted output, through the second MERGE, and back around to the acknowledgment on the sequencer. Only then is the second signal allowed through the sequencer, precluding any possibility of contention for the first MERGE. If the second signal has not arrived yet, the sequencer waits for it and then lets it through. Otherwise it goes through immediately. The second signal reaches the TOGGLE and triggers the undotted output, which sends an output signal on x . This signal also feeds back through the second MERGE to acknowledge the sequencer again, allowing the cycle to repeat for two more inputs. Hence, a single x output is observed for each pair of inputs.

2.4.3 Implications of this solution

This exercise draws to a close at last with this design. Unlike the previous attempts, there is nothing to go wrong with this one. No combination of wire or component delays, however malicious or well coordinated, can conspire to cause this circuit to malfunction, assuming only that they are finite and non-negative. This condition is the intuitive essence of delay insensitivity, which can be appreciated

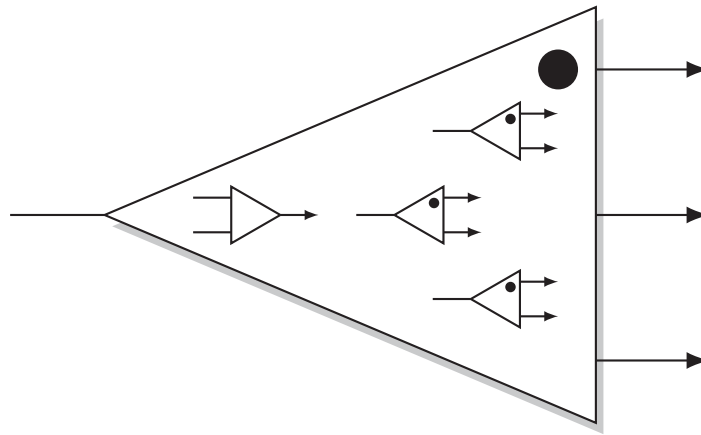


Figure 2.11: A three way TOGGLE can be made from the ordinary TOGGLE and a MERGE by connecting the wires with a little ingenuity on a TOGGLE-shaped breadboard.

fully at this point in view of the alternatives considered along the way (*i.e.*, synchronous, bounded delay, NCL, SI, QDI, fundamental mode, burst mode, and extended burst mode).

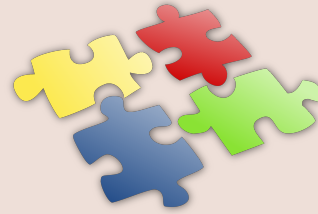
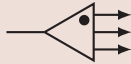
Although a DI implementation of a majority gate is shown to be possible by this exercise, this implementation is more costly than what could be achieved in a no-holds-barred transistor level design verified by analog simulation [24]. If majority gates are a known bottleneck for the performance or area of project, investing in a custom designed primitive library module to that effect may be justified.

A few final observations about this exercise set the agenda for the rest of the book.

- Each of the four solutions including the incorrect ones has been well supported by a hand-waving argument. Some method of formal verification would be helpful to avoid misplaced confidence.
- At each iteration, the circuit becomes less intuitive, leading ultimately to a solution justifiable only in hindsight. Without automated synthesis or at least some systematic methodology, DI design remains a black art.
- It should not be standard operating procedure to invent new DI primitives along the way to every solution. A finite set of primitives sufficient for any purpose should be sought.

Puzzle page

1. What else could go wrong in [Figure 2.5](#)?
2. What happens next in [Figure 2.7](#)?
3. A generalization of the TOGGLE primitive has three outputs instead of two.



The first time it receives a signal, the 3-way TOGGLE acknowledges it by the dotted output. The second time, it uses the middle output to acknowledge the signal, and the third time, the bottom output. The cycle repeats after that. Improve a 3-way TOGGLE by wiring three ordinary TOGGLE primitives and a MERGE to the external terminals shown in [Figure 2.11](#) so as to leave a user none the wiser.

4. A change in the manufacturing process has made the MERGE primitive ten times faster than the TOGGLE primitives. Modify the 3-way TOGGLE design above as needed to compensate for the difference. (hint: This question is a kōan.)
5. What extra feature does the circuit in [Figure 2.10](#) have that is not strictly required by the 2-of-3 majority gate specification? (hint: What happens when there are three concurrent input signals?)
 - a) Should we conclude that the circuit does not meet the specification? Why or why not?
 - b) What simpler primitive than a sequencer could enable a more efficient design that just meets the specification without exceeding it? (hint: [Figure 13.7](#))
6. Modify the circuit in [Figure 2.10](#) to make
 - a) a 2-of-4 majority gate
 - b) a 3-of-4 majority gate.

Assume the availability of a component library equipped with a sequencer, a TOGGLE, and a MERGE having any required number of inputs or outputs.

7. A software manager wants a sequencer to give priority to a request R_1 whenever R_1 and R_2 occur simultaneously, but to grant them in the order received otherwise. What issues should the hardware team raise in their next meeting?

Nothing is more practical than a good theory.

Ludwig Boltzmann

CHAPTER

3

THE LAY OF THE LAND

Taking a DI circuit design from an initial concept to a working implementation is generally too big of a job to do in one step because there are too many details to remember and too many ways for it to go wrong. Like an experienced caravan leader, the competent designer stops at one or more intermediate destinations to confirm that everything is in order up to that point. There may be a choice of routes, with some better than others depending on the circumstances [60].

For the DI designer, each intermediate destination is a particular concrete representation of the design. Each of these representations completely determines the design, but emphasizes some aspects of it over others, both as an aid to intuition and to facilitate automated checking of desired properties. Furthermore, the transformations from one representation to another are also automated to avoid introducing errors *en route*.

The management of complexity through abstraction and sound engineering methodology is nothing new, but the great strength of DI design in this regard is its capacity to benefit from a set of well honed end-to-end algorithmic tools that do not readily extend to asynchronous design styles with stronger delay assumptions. For example, the question of whether a supposedly improved version of a circuit is a compatible replacement for the original (*i.e.*, a **refinement** of it) can be answered automatically for DI circuits by methods described here without need of manual intervention or *ad hoc* proof techniques. QDI and SI designers faced with similar questions would need to look further afield (*e.g.*, [32, 72, 232]).



The balance of this chapter gives an overview of the ways DI circuits may be specified and understood, along with some related informal commentary on specific areas as a prelude to a more rigorous development in Part II. The remaining aspects of a normal work flow, involving methods for transformation, optimization, verification and review, are described in [Chapter 4](#).

3.1 Overview

A graph of the main concrete representations of interest and the known transformations among them is shown in [Figure 3.1](#). This diagram is not a flow chart of steps to be followed in sequence, but rather a map of the relationships among entities a designer may find useful for establishing confidence in a design. Each solid arrow corresponds to a transformation algorithm discussed somewhere in this book, and each box corresponds to a concrete representation.

The concrete representations are broadly divided between two groups, either human-writable or compiler generated. The compiler generated group further includes a group of finite automata in the familiar sense of recognizers for regular languages [115].

- The human-writable group constitutes the entry points for a designer to construct a specification using a text editor or visual development environment.
- The compiler generated representations are not expected to be convenient to construct manually, but may provide useful feedback to a designer in graphical form or through automated test results.

Some details are omitted from the diagram to avoid unnecessary clutter. An arrow from most representations back to themselves could also be included because many of them permit semantics preserving transformations to more efficient or compact forms. Certain intermediate representations of no independent interest are also omitted.

Although it is not formally part of the theory as such, a box corresponding to the target technology is also included in the diagram to show the most likely technology mapping path (from the flat netlist representation). The nature of the target is beyond the scope of this book because it is technology dependent and not specific to DI design. Nor is the dotted arrow discussed further in this book, although it might correspond to an automated transformation as well. For an FPGA target, it might encompass the final placement and routing phases. For a custom VLSI target, it might involve layout generation. For some unknown future technology, it might be whatever the reader envisions.

3.2 The process model

Along with this assortment of concrete representations, it is also convenient to keep in mind an abstract concept of a DI circuit independent of any of them. As noted in [Chapter 2](#), an abstract model of DI circuits as a function that statically maps inputs to outputs is inappropriate and unworkable, however useful such a model may be for Boolean networks. Fortunately, there is a comparable abstraction sufficiently expressive to capture the essence of a DI circuit in quite a natural way, known as a *process*. There is a rich and sometimes impenetrable literature on this subject, but fundamentally processes are a simple and readily approachable idea.

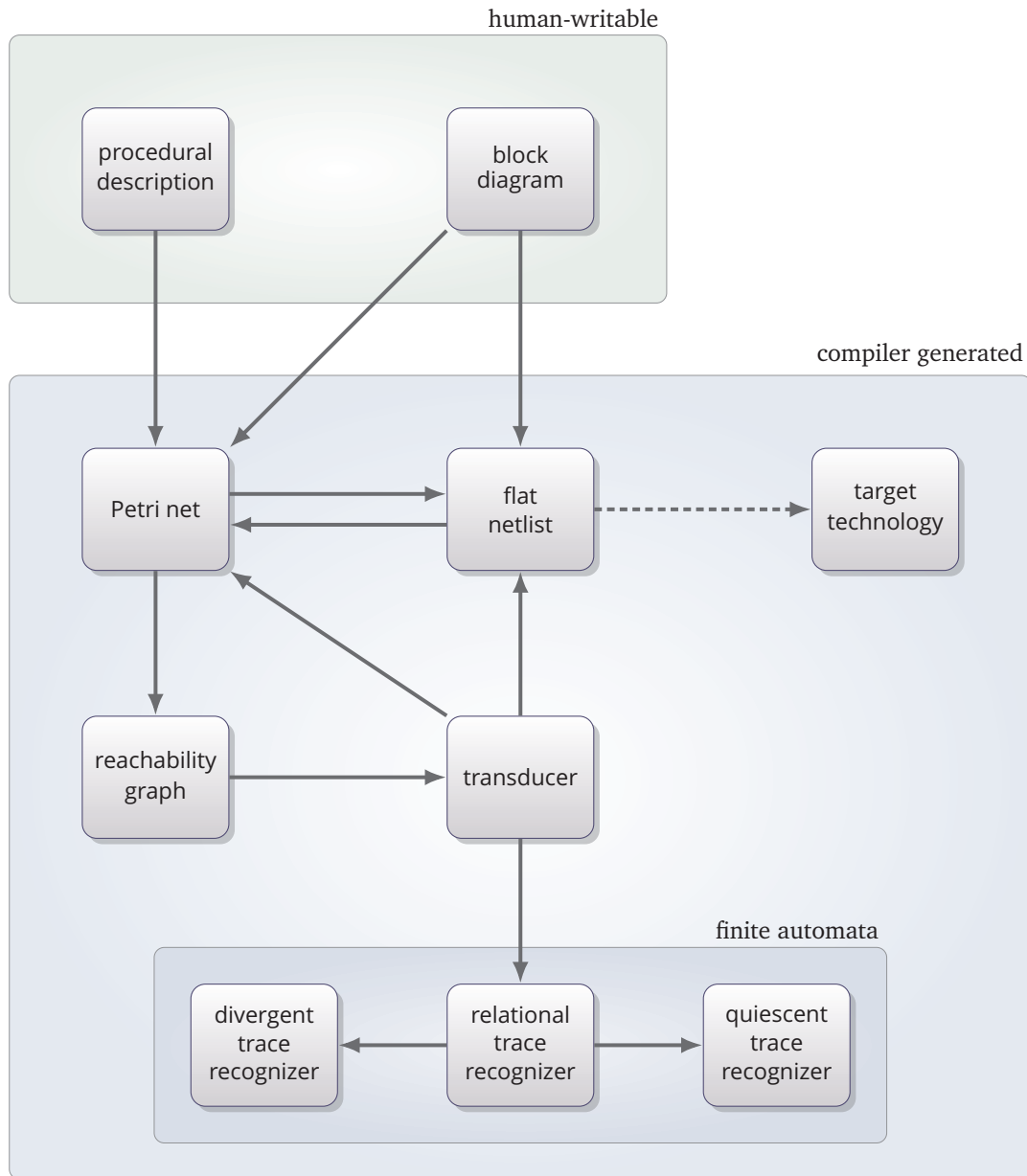


Figure 3.1: automated transformations among DI circuit representations

3.2.1 Process concepts

A process can be understood informally as an extensional agent (or in engineering terms, a “black box”) that interacts with its environment according to a specified protocol. The notion of a protocol can be made precise for the most part by identifying it with the (usually infinite) set of sequences of discrete, atomic actions deemed to be compatible with it.

- In any sequence of actions, some are taken by the environment and some by the process being modeled.
- The actions of a process associated with a DI circuit and its environment are the manifestations of signals on the input and output terminals.
- Input signals to the circuit correspond to actions taken by the environment, and outputs from the circuit correspond to actions taken by the process.

3.2.2 Generality

By judiciously allowing some sequences into the set and excluding others, we can construct protocols encoding arbitrarily complex constraints on the future actions of either the process or its environment due to previous actions. Concurrency is expressed by admitting a multiplicity of similar sequences differing only in the ordering among concurrent actions. The process abstraction is therefore an extremely powerful one, subsuming functions, state machines, and many general purpose computational models, all with remarkable conceptual economy. In addition, a process model is more conducive to a satisfactory account of communication among multiple agents.¹

3.2.3 Environments

The concept of an environment is essential to the understanding of a process, far more so than in the case of other computational models. Our cultivated agnosticism regarding the inner workings of a process allows only the interface with its environment as the recognized venue for its activity. For the process associated with an individual component within a circuit, the environment is the rest of the circuit. For two interacting circuits, the environment of each is the other circuit. For an isolated circuit, the environment is the user. Similarly to people, a process that thrives in one environment could falter in another. It is therefore essential to consider compatibility with an environment when assessing correctness.

3.3 Block diagrams

As shown in [Figure 3.1](#), the procedural description and the block diagram are two complementary ways for a designer to describe a DI circuit. Block diagrams are a generic technique widely used in many areas of engineering, whereas the procedural description outlined here is specific to DI circuits. Any project may involve a combination, depending on the nature of the specification and preferences of the designer. Typically procedural descriptions are suitable when the flow of control is more cumbersome to describe than the dataflow, and block diagrams are more appropriate in the alternative. Block diagrams are the topic of the remainder of this section, and the procedural description is introduced in [Section 3.6](#).

¹Fairness properties and progress obligations can be problematic, but the rest will do nicely for now.

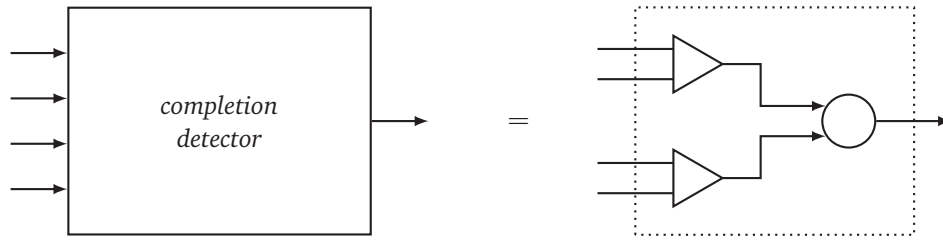


Figure 3.2: A subsystem in a block diagram is encapsulated in an opaque block identified by a descriptive name (for example, a “completion detector”) and defined by its contents.

3.3.1 Notation

A block diagram is rendered as system of blocks interconnected by arrows. Although block diagrams are often used informally in other disciplines, we will insist on the block diagram of a DI circuit having rigorous semantics. Each arrow corresponds to a wire in the circuit, and each block represents a constituent part of the circuit, best envisioned as a process. Circuit diagrams such as [Figure 2.2](#), [Figure 2.8](#), and [Figure 2.10](#) are block diagrams in which the blocks are either primitive DI components or hidden combinations of them depicted as a unit, such as the sequencer in [Figure 2.10](#). Because blocks can be made up of multiple primitives, there might be more than one way to partition a given circuit into blocks. It is left to the designer’s discretion to designate the block boundaries, but the intent should be to decompose the design into parts serving individually meaningful purposes.

3.3.2 Methodology

A basic illustration of how a designer might use block diagrams wisely is sketched in [Figure 3.2](#), which shows a dual rail completion detector.² To transmit two bits concurrently, two channels are used, requiring two pairs of wires. To confirm the arrival of both bits, a circuit on the receiving end needs to detect a signal from either wire of both pairs. The reader should take a moment to convince himself or herself the circuit on the right of [Figure 3.2](#) meets this need. Many instances of a completion detector might be deployed in a large design. To avoid revisiting the same train of thought repeatedly and to avoid cluttering the diagram, the designer abstracts the concept of a completion detector as a single block, shown on the left of [Figure 3.2](#).

Block diagrams are most effective when developed hierarchically, as illustrated in [Figure 3.3](#). Rather than constructing every block starting only from primitive components, blocks can be built by combining smaller blocks into larger ones. Hierarchical block diagrams facilitate a certain flexibility in design styles.

- Some designers may prefer a *top-down design*, whereby a course version of the diagram consisting of just the main blocks is constructed initially without advance knowledge of their precise semantics. Each of these blocks is successively elaborated until the specification is complete.

²In a dual rail coded channel, a stream of 1’s and 0’s is carried by two wires. A signal sent on one of the wires communicates a 0, and a signal on the other communicates a 1. [[14](#), [293](#)].

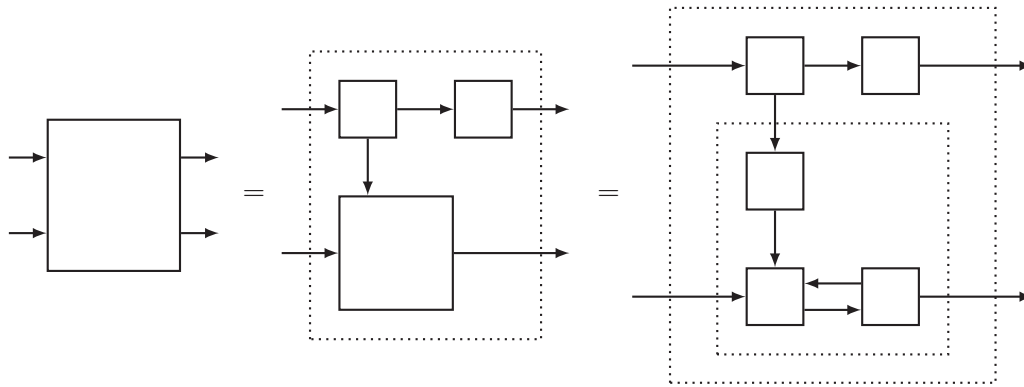


Figure 3.3: Blocks in a block diagram can be nested as an aid to abstraction.

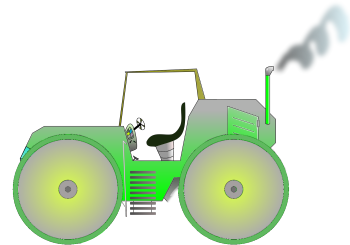
- Other designers opt for a **bottom-up design**, whereby blocks performing simple operations are constructed first, and subsequently assembled into larger configurations until the whole system is specified.
- A lesser known alternative approach, sometimes called **edge-in design**, consists of a concurrent top-down and bottom-up design. Letting the big picture and the detailed view inform each other, the designer is sure to arrive at a viable result when they meet in the middle.

Each of these design styles has its advantages and disadvantages. Mistaken assumptions tend to be discovered later in top-down designs than in the others. Bottom-up designs are more prone than the others to prematurely completed work needing to be discarded or revised subsequently. An edge-in design can be difficult to plan or to delegate among a team, and may demand multi-disciplinary expertise from an individual designer.

3.3.3 Flattening

By recursively substituting each composite block in a block diagram with the network of constituent blocks it represents, a block diagram can be transformed automatically to a flat netlist (per [Figure 3.1](#)). The flat netlist representation contains only primitive components and their connecting wires, having been stripped of any hierarchical or symbolic information meaningful to the designer.

This transformation might be made when no further modifications to the design are anticipated, as the last step before a final technology mapping phase. While the route from the block diagram to the target technology via flat netlists is agreeably short, it bypasses any formal verification and could allow errors to remain unnoticed until the product is finished, which would be expensive to correct by then. A more prudent course might allow for a detour through safer territory, as we explore from this point onwards through [Chapter 4](#).



3.4 Towards a process semantics

Whatever the methodology, block diagrams are helpful for keeping a large design organized, and they easily lend themselves to automated tool support. However, without some sort of a semantics underlying them, they are not much of an advancement over drafting circuit diagrams on paper. We need some way of establishing that a circuit will perform as intended. To this end, there are at least three basic characteristics any worthwhile semantics should have.

- It should be **prescriptive**, meaning that it can be used to express the designer's intention about what a circuit should do in some simpler or more compact manner than the finished circuit design itself.
- It should also be **descriptive**, in the sense of enabling an unambiguous automatic derivation of what a given circuit really does.
- Furthermore, it should be **analytical**, in that it equips a designer to compare the intended with the realized behavior, also preferably by automatic means.

3.4.1 Trace structural composition

It would be straightforward to attempt a process oriented semantics for block diagrams fulfilling these three desiderata by specifying a chosen set of DI primitives as processes (according to some formalized the notion of a process), and then by specifying a semantics for composition among processes that corresponds to physical connection among blocks. However, pursuing this course too naively could lead to unforeseen difficulty. The typical construction goes something like the description below, and a deconstruction follows in [Section 3.4.2](#).

1. Ignoring cosmetic differences among various sources for the moment [55, 72, 80, 128, 171, 262, 288, 294], we formalize a process as a **trace structure** (A, T) with a finite **alphabet** A partitioned into disjoint input and output alphabets I and O , and a set $T \in A^*$ of **traces**, containing finite sequences of symbols from the alphabet.
2. To define composition, suppose two trace structures $X = (A_X, T_X)$ and $Y = (A_Y, T_Y)$ have input alphabets I_X and I_Y , respectively, and output alphabets O_X and O_Y , respectively (*i.e.*, $A_X = I_X \cup O_X$ and $A_Y = I_Y \cup O_Y$). Their composition $Z = (A_Z, T_Z)$ should interact with its environment the same way the circuits represented by X and Y would interact jointly with their environment if they were thrown together somehow.
 - a) The inputs and outputs exposed by Z therefore should be the same as those those exposed by X and Y for the most part, except in the case of an input to X being an output from Y or *vice versa*. Any terminal that is an input to one of X or Y and an output from the other becomes neither an input nor an output of their composition Z , but an internal connection hidden from the environment instead. Hence we define A_Z as the union of input alphabet I_Z and output alphabet O_Z , where they satisfy

$$I_Z = (I_X - O_Y) \cup (I_Y - O_X) \quad (3.1)$$

$$O_Z = (O_X - I_Y) \cup (O_Y - I_X). \quad (3.2)$$

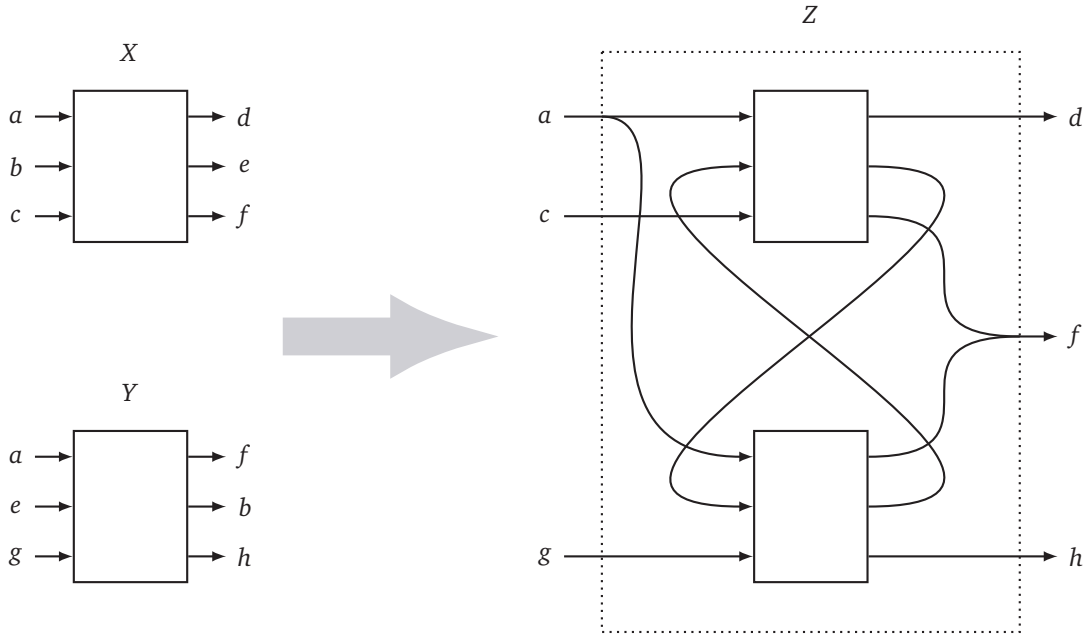


Figure 3.4: In a trace structural composition, common inputs a are forked, common outputs f are merged, and signals b and e that are both inputs and outputs are connected internally and hidden.

- b) The traces T_Z of the composition Z are given by the interleaved traces of X and Y running concurrently and synchronized with each other by the now hidden symbols, if any (sometimes called the **weave** of T_X and T_Y). The following definition captures this idea more formally.

$$T_Z = \{t' \in A_Z^* \mid \exists t \in (A_X \cup A_Y)^*. t \uparrow A_X \in T_X \wedge t \uparrow A_Y \in T_Y \wedge t' = t \uparrow A_Z\} \quad (3.3)$$

The notation $t \uparrow A$ for a trace t and an alphabet A means the trace obtained by deleting all non-members of A from t , also known as the **projection** of t onto A . In [Equation 3.3](#), the component processes X and Y participate in a trace t with each other, but the environment sees only t' , which is t with the hidden symbols suppressed.

If the equations above seem complicated, it may be helpful to regard them simply as a precise way of describing the situation depicted in [Figure 3.4](#) in general terms. Here we have two trace structures X and Y whose input alphabets I_X and I_Y are respectively $\{a, b, c\}$ and $\{a, e, g\}$, and whose output alphabets O_X and O_Y are respectively $\{d, e, f\}$ and $\{f, b, h\}$. What should it mean to compose X and Y when these alphabets intersect? For better or worse, the theory stipulates that the shared input a is forked to both, the shared output f is merged from both, and the signals b and e are connected between them and hidden from the environment. The combined process is whatever results from X and Y running normally to interact with each other and the environment, but with b and e no longer visible externally.

3.4.2 Deficiencies of a naive trace structural composition

Although this theory may look good on paper, its shortcomings yield to a moment of scrutiny, starting with the concept of a trace set.

- For any process of practical interest, the trace set T is infinite, and hence impossible to exhibit directly in any rigorous sense or to manipulate without further embellishment to the theory, such as regular expressions, process algebras, or something more imaginative.
- If a trace set is interpreted to represent all acceptable interactions between a process and its environment up to a point in time, it fails to express a crucial distinction. A trace set $\{a, ab, aba, abab \dots\}$ could represent a process that always answers an input of a with an output of b , or could just as well represent one that chooses non-deterministically to refuse all further communication at any moment. If this point seems pedantic because the latter obviously would never be intended or expected in practice, then it is arguably even more important for a semantic model to be able to raise the alarm when it happens. In the jargon of the trade, the trace set is said to be unable to represent *progress obligations*.
- With this interpretation of a trace set, it is also impossible to construct a useful refinement relation (for checking whether one circuit can be a compatible replacement for another).
- A careful reading of [Equation 3.3](#) reveals that any shared inputs between circuits in a composition are effectively broadcast to both, but we would soon find out the hard way that some form of arbitration or alternation between them would be far more useful in practice, and it boggles the mind to contemplate generalizing [Equation 3.3](#) accordingly.

To top off all of these issues, the definition of composition in [Equation 3.3](#) suggests no efficient procedure for computing it. This style of semantics as it stands therefore is neither prescriptive, nor descriptive, nor analytical in the sense described at the beginning of [Section 3.4](#).

3.5 Petri nets

A more holistic diagnosis of the present difficulty might be that the semantic gap between block diagrams and processes either precludes a direct correspondence between them or incurs a risk of oversimplification. To make the situation more manageable, a Petri net can be used as an intermediate representation. As a graphical formalism, Petri nets lend themselves to easy translation from a block diagram. On the other hand, a Petri net induces a process model in a straightforward way that is both finitely describable and more expressive than a trace set alone.

3.5.1 Notation and conventions of Petri nets

A substantial body of literature has accumulated around Petri nets, but they are based on fundamentally simple ideas. A Petri net is envisioned informally as a collection of vertices, with some vertices connected to others by arcs. There are two kinds of vertices: transitions and places. Transitions can be connected only to places and places can be connected only to transitions. Every arc is considered to have a direction *from* some vertex *to* some other one.³

³Some sources define Petri nets as directed bi-partite graphs, but a construction in terms of adjacency matrices is equally valid, and the choice between them is arguably a matter of implementation [201].

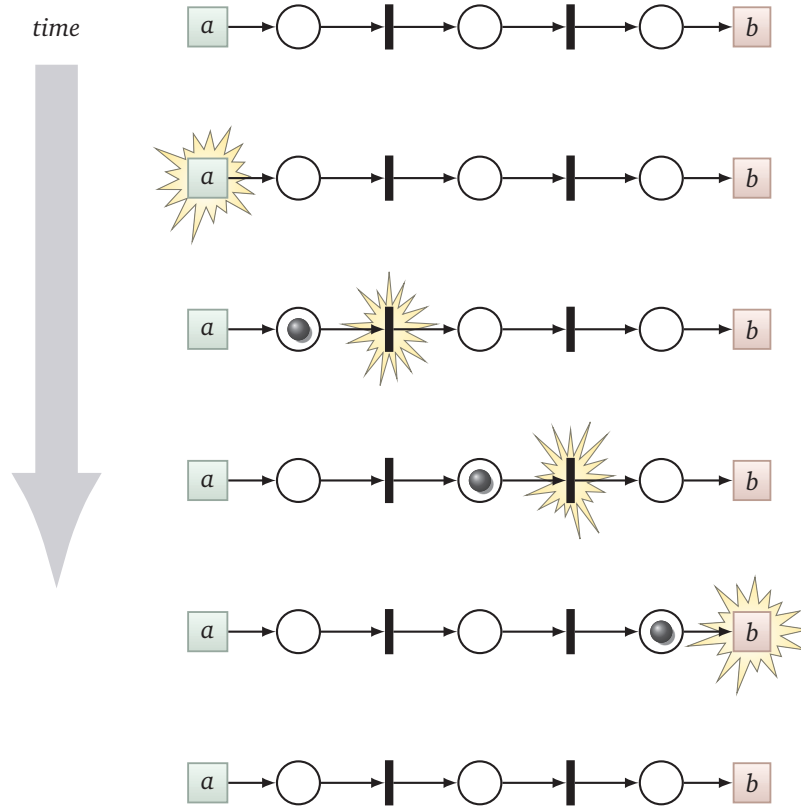


Figure 3.5: animation of six snapshots of a simple Petri net, showing a token absorbed and/or emitted each time a transition fires, until the Petri net reverts to its original tokenless condition

Depiction

Conventionally a Petri net is drawn with circular places and rectangular transitions, and with the arcs depicted as arrows, as shown in Figure 3.5. For our purposes, some transitions are labeled as inputs or outputs, with inputs in green and outputs in red, and the remaining transitions are anonymous. We think of inputs and outputs as tangible entities capable of exchanging signals with their environment. The rest of the Petri net exists only as a drawing on paper or pixels, but explains how the behaviors of the inputs and outputs are related.

Operation

To express this relationship, the Petri net is marked with movable tokens depicted as black balls, like playing pieces in a board game, which advance across transitions or rest in places according to these simple rules.

- Whenever every incoming arc on a transition is connected to a place marked with a token, that transition is said to be *enabled*.



Figure 3.6: Petri nets clearly distinguish between two processes with similar interfaces but different progress obligations. The left Petri net always acknowledges an input on a with an output on b , which the right one need not.

- To avoid any misunderstanding, we stipulate that transitions with no incoming arcs are always enabled, and any transition that is not enabled by either of these criteria is **disabled**.
- Whenever any transitions are enabled, the next thing to happen will be for exactly one of the enabled transitions to **fire**.
- When a transition fires, every place connected to any of its incoming arcs loses a token, and every place connected to any of its outgoing arcs gains a token.
- The rearrangement of tokens due to a transition firing causes transitions to be newly enabled or disabled accordingly.

An example of a Petri net

To convey a feel for the way a Petri net works, Figure 3.5 depicts an animation of six possible consecutive states for a Petri net consisting of a simple linearly connected sequence of transitions and places. In the first state shown at the top, only the input a is enabled. Eventually it must fire, as shown in the next image below. In so doing, it deposits a token in the first place on the left, which enables the firing of the anonymous transition to its right. As a result, the token is evacuated from the left place and deposited in the middle, enabling the next anonymous transition. When that one fires, the token moves to the right again, enabling the output b , which then fires and removes the token.

3.5.2 Expressiveness of Petri nets

There is much more to be said about Petri nets, but this incomplete description will suffice for now as we return to some of the questions that motivated their introduction. The cumulative effect of the Petri net in Figure 3.5 is to constrain tokens to propagate from left to right, causing every firing of a to be followed inevitably by a firing of b , so that the set of possible firing sequences (projected onto the set of observable transitions) spells out the trace set $\{a, ab, aba, abab \dots\}$.⁴ A charitable reader might grant that Petri nets can be constructed to express a great variety of trace sets, but to what end?

⁴subject to a provision restricting consideration to so called 1-safe firing sequences, to be described on page 53, and neglecting the empty trace ϵ

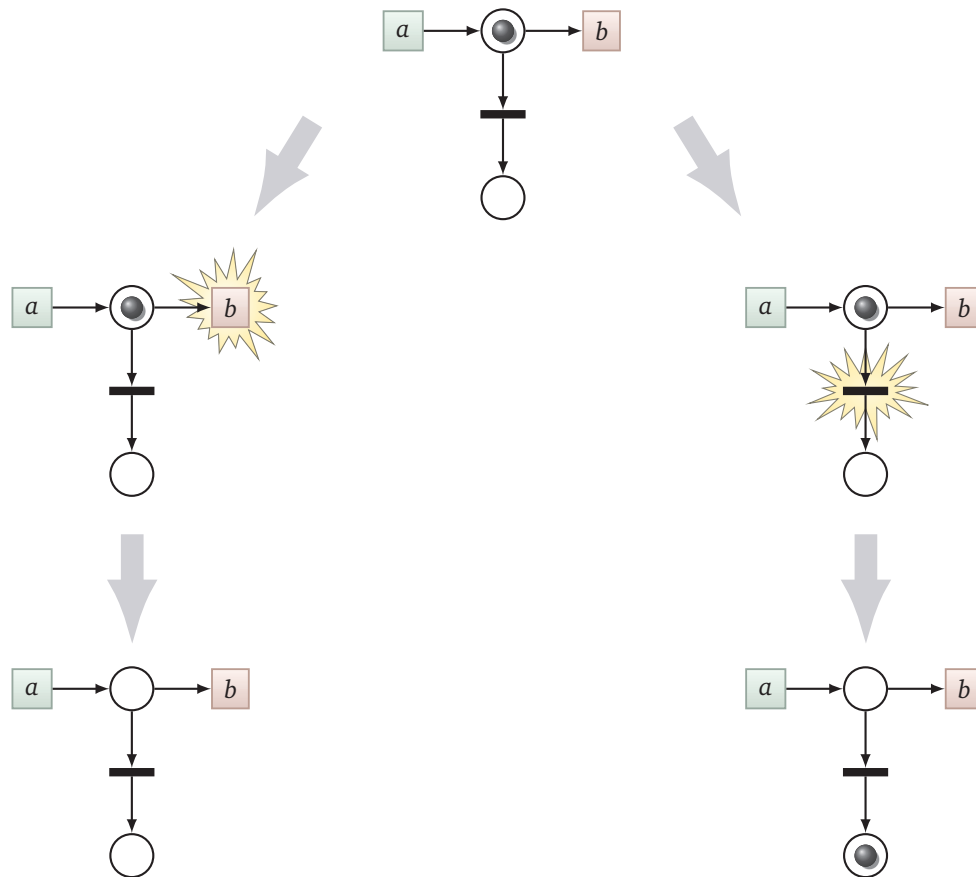


Figure 3.7: Petri nets model non-determinacy. From the marking above, either course may be taken, depending on which transition fires next.

Part of the answer can be seen in [Figure 3.6](#), which shows two Petri nets side by side. Similarly to the Petri net in [Figure 3.5](#), each of these Petri nets induces a trace set $\{a, ab, aba, abab \dots\}$. However, the one on the left always acknowledges an input of a with an output of b , while the one on the right need not. A walk through the latter shown in [Figure 3.7](#) demonstrates the reason. An input of a deposits a token in the upper place, which enables both the output transition b and the anonymous transition. The choice of the next transition to fire is non-deterministic. If the anonymous transition fires, no output on b is received by the environment. Hence, a Petri net model expresses progress obligations or lack thereof in a way that eludes trace sets alone, at least as they are conceived up to this point (*cf.* [Section 3.4](#)).

3.5.3 Compositionality of Petri nets

Another way Petri nets come out ahead of a naive trace structural semantics for block diagrams is by making composition easier to implement than [Equation 3.3](#). That is, a Petri net model for a network

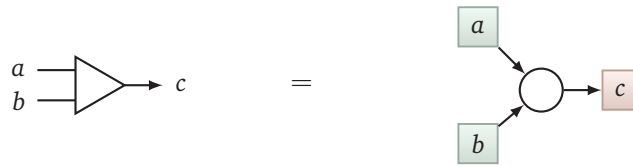


Figure 3.8: Petri net model of a MERGE primitive

of connected circuits is derivable from the models of the individual circuits without any complex calculations required. Although we have yet to take proper account of all DI primitives and their Petri net models, a sneak preview of a simple but important case supports this claim.

Model of a primitive

The Petri net model for a MERGE primitive is shown in [Figure 3.8](#). Why is the model as shown and not anything else? The firing of either input a or b would deposit a token in the place, which would enable the output c to fire subsequently, as expected of a MERGE. Concurrent inputs to a MERGE are prohibited, as explained on [page 35](#), and the Petri net model reflects this requirement as well. If the input transitions a and b both fired without the output c firing inbetween, the place would be filled with two tokens. We can and will consider any marking with multiple tokens in the same place a sign that something has gone wrong, and call it an *unsafe* marking. Any firing sequence that could lead to an unsafe marking is outside the range of behavior the Petri net expresses. Hence, the Petri net model as shown tells us everything there is to know about a MERGE.

Model of a composite circuit

Although it is simple enough to construct a Petri net model for a primitive component such as a MERGE and to justify it by reasoning as above, it would be unproductive to pursue a similar analysis for every circuit we design. For example, it should be a consequence of the semantics and not a matter of hand waving or *ad hoc* axiomatization that a tree of connected MERGE primitives behaves as a multi-way MERGE.

How well the Petri net model succeeds in this regard can be judged from [Figure 3.9](#). As shown in the figure, composition of Petri nets consists of replacing each pair of similarly labeled input and output transitions with a single anonymous transition. This operation is the graphical analog of [Equation 3.1](#) and [Equation 3.2](#). Moreover, the trace structure induced by this construction corresponds precisely to the weave of the two constituent trace structures as given by [Equation 3.3](#), not to mention the behavior of the three way MERGE circuit it represents. That is, the firing of any input a , b or d leads directly or indirectly to the firing of the output e . Any consecutive (or simultaneous) input firings without an intervening output can cause multiple tokens to accumulate in the same place and are therefore prohibited.

Readers having a certain mathematical disposition may find it helpful to view [Figure 3.9](#) as an example of a *commutative diagram*, which in this form asserts an essential sanity check on any semantic model aspiring to a property known as *compositionality*. Simply stated (maybe a little too simply for some tastes), this property requires the semantics of any compound entity to be fully

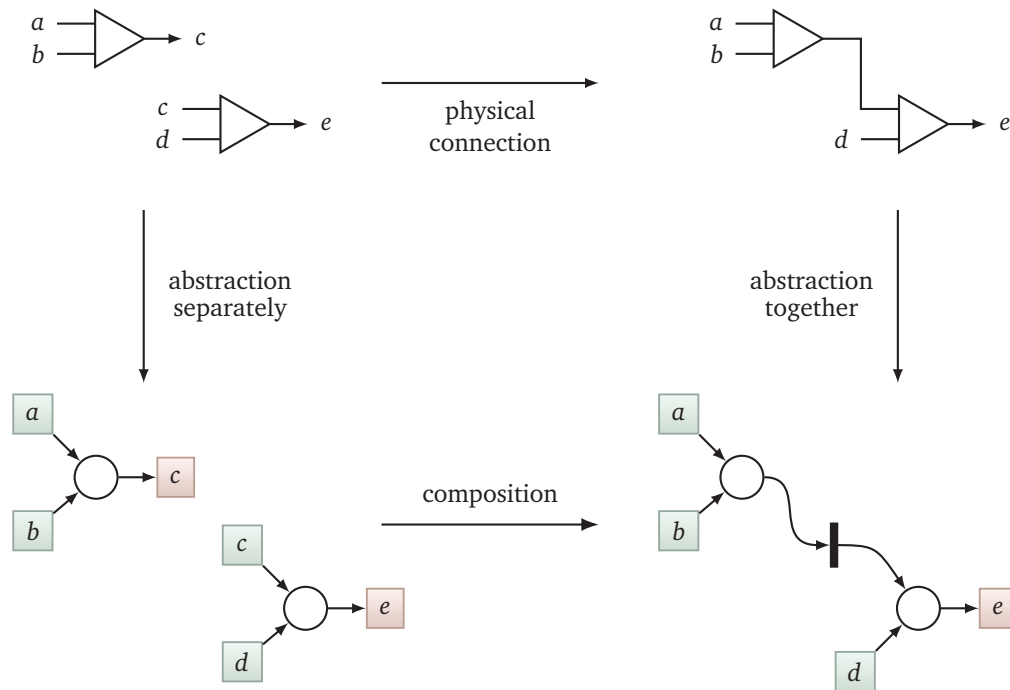


Figure 3.9: A Petri net model accurately describing a network of interconnected circuits is trivial to construct from their respective individual Petri net models by connecting them the same way. This trick does not work with state machines.

determined by the respective semantics of its constituent parts, disallowing any dependence on context, scope, environment, phase of the moon, *etc.* [276]. The diagram expresses this idea by showing that obtaining the Petri net models for two circuits independently and then composing the models leads to the same result as connecting the circuits first and then finding a Petri net model for the connected circuit.

3.5.4 Limitations of Petri nets

Using Petri nets as an intermediate representation alleviates some of the shortcomings of a naive trace based circuit semantics, but not all. It would be fair to claim that it meets the descriptive criterion mentioned in [Section 3.4](#) because Petri nets readily capture the behavior of arbitrarily complex circuits with ease. However, the Petri net representation is not convenient in itself for evaluating refinement or equivalence between circuits (not least because Petri nets with different innards could exhibit the same outwardly observable behavior), and hence is less effective as an analytical semantic model. Nor is the Petri net representation profitably prescriptive. The effort required to construct a Petri net model manually to some non-trivial behavioral specification is comparable to that of manually designing a



Type	Mnemonic	Description
$\mathbb{T} \rightarrow \mathbb{D}$	get put	receive a signal send a signal
$(\mathbb{D} \times \mathbb{D}) \rightarrow \mathbb{D}$	seq par alt env	do one after the other do both concurrently do either but not both do no more than required to interact
$(\mathbb{D} \rightarrow \mathbb{D}) \rightarrow \mathbb{D}$	fix	act as the solution to a recurrence

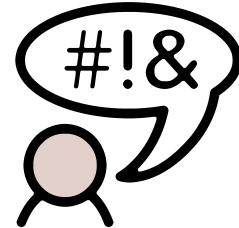
Table 3.1: Combinators defined as functions operating on signal terminals of type \mathbb{T} and DI processes modeled by Petri nets of type \mathbb{D} are the primitive operations of a procedural circuit description formalism.

circuit to that specification.

Keeping in mind that the semantic framework outlined in [Figure 3.1](#) employs various mutually fungible concrete representations, we need not count these limitations on Petri nets as serious obstacles. The need for a prescriptive semantics is met by block diagrams along with the procedural description to be explained next. The analytical aspects are discussed starting in [Chapter 4](#).

3.6 Procedural description

By an approach complementary to block diagrams also indicated in [Figure 3.1](#), a procedural narrative determines a target representation similarly to a software development work flow. While the ultimate target is hardware, Petri nets serve as an intermediate representation here as well, both to maintain a layer of technology independence, and to promote interoperability with block diagram specifications.



The translation is fairly straightforward. [Table 3.1](#) shows a small fixed set of combinators chosen to facilitate circuit specification. They operate on signal names and Petri net fragments to build them into useful arrangements and are suitable as the core primitives for the back end of a compiler. A front end could expose these operations in a domain specific language either by leveraging an established DSL framework [[66](#), [209](#), [215](#), [296](#), [297](#)] or a parser generator tool [[74](#), [190](#), [217](#)], or alternatively just by way of a no-frills hand-coded recursive descent parser [[7](#)].

3.6.1 Combinator examples

Without delving into the formal semantics at this point, we can get an idea of the translation method by exploring some of the combinators listed in [Table 3.1](#) informally. By design, they always generate Petri net fragments with initially marked places having no incoming arcs. The combinators also designate certain places as final, which have no outgoing arcs. The final places are where the tokens in a Petri net fragment should end up if they start in the initial places and follow the rules from [page 50](#). The final places are depicted with shading to suggest a golf course hole or any comparable potential well in the page surface.

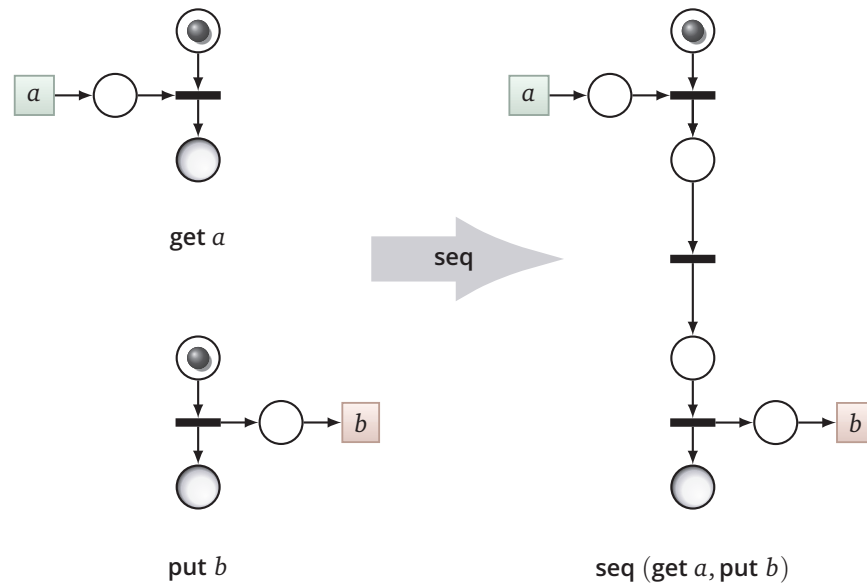


Figure 3.10: Petri net fragments generated by the input, output, and sequential composition combinators, shown with the initial places at the top and the final places (indented) at the bottom

Sequential composition

The definitions of the **get**, **put** and **seq** combinators follow intuitively from these invariants as shown in Figure 3.10. The initially marked place is shown at the top of each Petri net fragment, and the final place at the bottom. For these examples, it is helpful to imagine a thread of execution carried by downward flowing tokens, and signals carried by tokens flowing from left to right. A Petri net fragment of the form **get a** accepts a single input signal of *a* and then reaches a terminal marking, whereas **put b** sends a signal by firing an output transition *b* and then terminates.⁵ The sequential composition of two Petri net fragments is obtained by connecting the final places of the first to the initial places of the second through a newly created transition, and unmarking the latter. In this way, the second Petri net fragment is blocked from executing until the first one finishes.

Parallel composition

Parallel composition is simpler than sequential composition because there is no need for any blocking. Given two Petri nets with no inputs or outputs in common, the **par** combinator creates a Petri net containing both in their entirety as disconnected components. If an input to one is an output from the other, the input and output are fused and anonymized (cf. Figure 3.9). It can also be useful to combine two Petri net fragments where the same signal is an input to both or an output from both. For shared outputs, the effect is the same as if the outputs were combined by a **MERGE**. When an input is shared, any signal on that input can be absorbed by either Petri net fragment. The Petri net model expresses arbitration between shared inputs in an obvious, natural way compared to a trace

⁵Actually, the output transition is enabled at the same time the final place is marked and therefore fires only subsequently, but this ordering of internal events has no observable consequences.

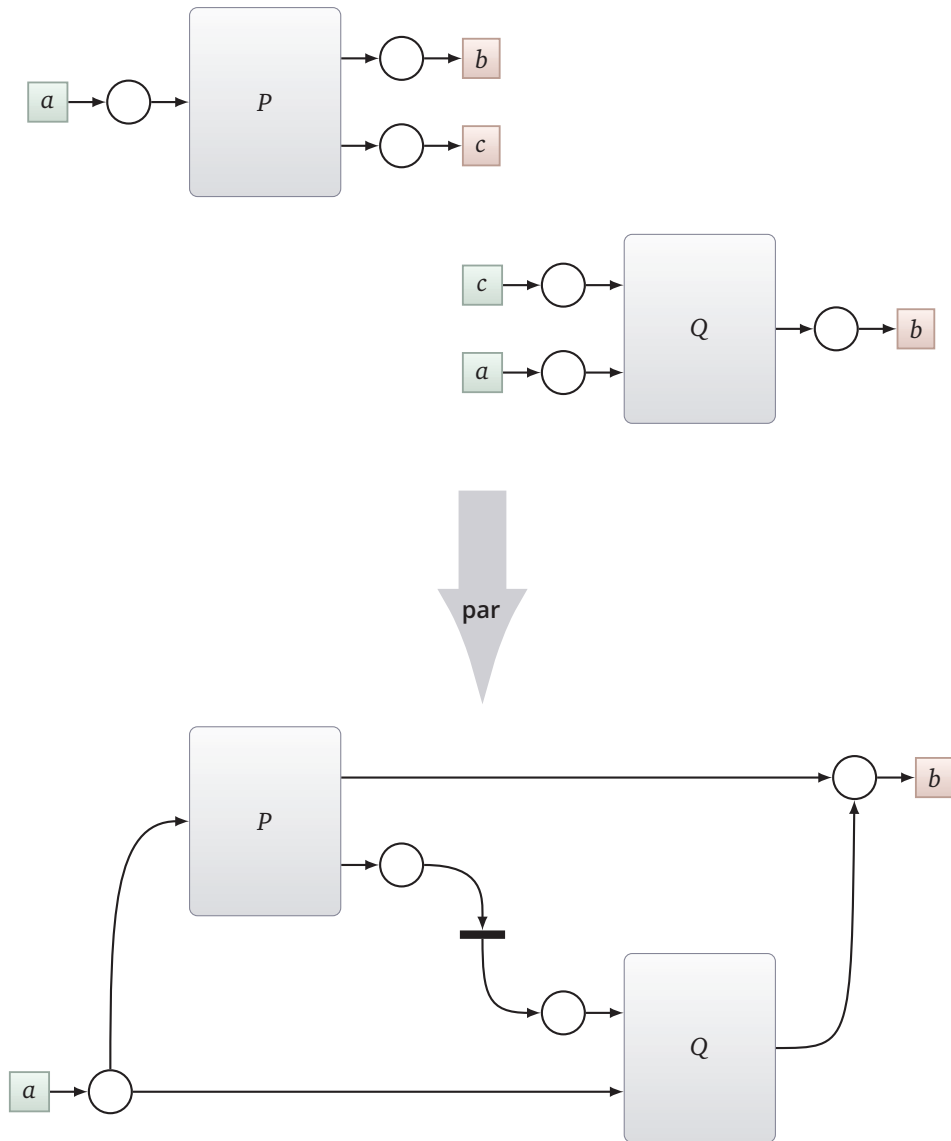


Figure 3.11: how the **par** combinator for Petri net fragments P and Q merges shared outputs b , arbitrates between shared inputs a , and hides a shared input and output c (cf. Figure 3.4)

structural semantics, dispensing with arbitration problem mentioned in Section 3.4. These three cases are illustrated in Figure 3.11. Similar conventions also apply to shared signals with the `seq` combinator.

3.6.2 Repetition

The sequential composition example in Figure 3.10 has only the finite trace set $\{\epsilon, a, ab, aba\}$ (where ϵ is the empty trace). No more than one output signal of b is ever emitted, and no more inputs of a after the second are acceptable. Obviously a circuit built to this specification is not useful in practice. In general, any useful circuit that works today should also work tomorrow. The `fix` combinator solves this problem by generating cyclic Petri net structures (whose details need not concern us at the moment) and could be used to implement loops or recursion in a higher level language. Alternatively, it could be exposed directly in a lightweight functional compiler front end making reasonable provisions for defining and referring to Petri net-valued functions in the language. Understanding how to use the `fix` combinator requires a quick sketch of the theory behind it.

Lambda abstraction

Probably the oldest and still unsurpassed programming language feature for expressing functions “anonymously” (*i.e.*, without having to name them) is **lambda abstraction**, especially if there is a need for functions that operate on other functions (so called **higher order functions**). Here is a crash course on lambda abstraction.

1. To define a function f that takes a number as an input and returns the successor of that number as a result, we write $f = \lambda x. x + 1$. This expression can be read as a recipe to compute the function f by substituting its argument for the variable following the λ (Greek letter lambda) in the formula following the dot, and returning whatever answer we get by evaluating the formula.
2. Higher order functions are made from nested lambda expressions. An expression like $h = \lambda k. \lambda x. x + k$ defines h as a second order function, in that it takes a number as an argument and returns a function as a result. To find $h(42)$, we substitute 42 for k (the outermost variable) and get $\lambda x. x + 42$, which is still a function, but now a first order function that adds 42 to its argument.

Specifying these rules properly and covering all of the edge cases leads to a formal system of symbol manipulation (*i.e.*, a calculus) known as **lambda calculus**, a foundational topic in computer science but not one central to this text.⁶

Fixed points

Using lambda abstraction, we could define a function f that takes a Petri net fragment as input, and returns a Petri net fragment as a result.

$$f = \lambda p. \text{seq} (\text{seq} (\text{get } a, \text{put } b), p)$$

⁶See [236] for an introductory tutorial, [18] or [110] for a canonical reference, and [3] for an advanced but lucidly written treatment with implications for programming language semantics.

There is nothing paradoxical or self-referential about f . Given any Petri net fragment x , it simply returns a Petri net fragment that waits for an input of a , outputs b , and then does whatever x does.

$$f(x) = \text{seq}(\text{seq}(\text{get } a, \text{put } b), x)$$

The next step to understanding the **fix** combinator is to ask the Zen-like question of what it would mean for a Petri net fragment w to satisfy $w = f(w)$. That is,

$$w = \text{seq}(\text{seq}(\text{get } a, \text{put } b), w)$$

Substituting the right hand side for w in the right hand side, we have

$$\begin{aligned} w &= \text{seq}(\text{seq}(\text{get } a, \text{put } b), \text{seq}(\text{seq}(\text{get } a, \text{put } b), w)) \\ &= \text{seq}(\text{seq}(\text{get } a, \text{put } b), \text{seq}(\text{seq}(\text{get } a, \text{put } b), \text{seq}(\text{seq}(\text{get } a, \text{put } b), w))) \\ &\vdots \end{aligned}$$

Such a w , if it existed, would be something like an infinite chain of alternating Petri net fragments of the form **get** a and **put** b , with the final place of each fragment connected to the initial place of the next by a transition, similarly to [Figure 3.10](#), and shared signals combined as in [Figure 3.11](#). In this way, it would express the behavior of a circuit that acknowledges an input a with an output b infinitely many times.

Given a function f and an argument w satisfying $w = f(w)$, we call w a **fixed point** of f (because transforming it by f leaves it unchanged). Fixed points are not always impossible in mathematics, but it is usually taken as part of the formal definition of a Petri net that it should be finite, so no such w can exist as a Petri net fragment. Nor would it be useful if it did, being impossible to write down or to simulate.

Pseudo-fixed points

Despite this technical issue, it is convenient to think of the **fix** combinator as one that constructs fixed points to order. To specify a circuit with infinitely repeatable behavior, we could write something like the following.

$$w = \text{fix } \lambda p. \text{seq}(\text{seq}(\text{get } a, \text{put } b), p) \tag{3.4}$$

Although w is not a fixed point of the function $\lambda p. \text{seq}(\text{seq}(\text{get } a, \text{put } b), p)$ in any formal sense, for practical purposes it exhibits the same observable behavior as the fixed point would. The **fix** combinator creates a cycle of arcs in the Petri net, but typically the cyclic structures are removed by compiler optimizations where possible. In combination with this and other optimizations, w could be transformed to the Petri net shown on the left of [Figure 3.6](#).

For functional programming *aficionados*, it is also straightforward to use the **fix** combinator to encapsulate common patterns of control flow, such as the following definition of an infinite loop constructor.

$$\text{loop} = \lambda p. \text{fix } \lambda f. \text{seq}(p, f) \tag{3.5}$$

This definition of a *loop* combinator allows the following simplification of [Equation 3.4](#), as the reader is in a position to verify.⁷

$$w = \text{loop } \text{seq}(\text{get } a, \text{put } b)$$

⁷hint: $\lambda p. h(p)$ is the same as $\lambda f. h(f)$

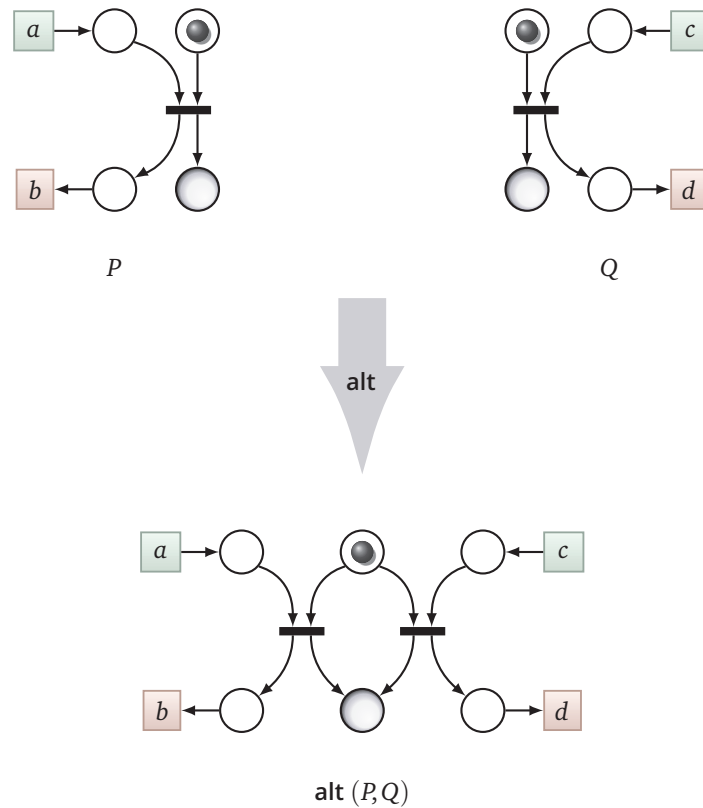


Figure 3.12: In the simplest case, the **alt** combinator fuses the respective initial and final places of its operands.

3.6.3 Conditional execution

In addition to doing things sequentially, concurrently, and repeatedly, circuits sometimes need to be selective about what they do. The **alt** combinator enables this behavior. Given two Petri net fragments, the **alt** combinator embeds them in a structure that allows one of them to execute and blocks the other. Because all Petri net fragments generated by the combinators in Table 3.1 have initial places with tokens in them, blocking one before it begins can always be done by evacuating the initial places.

Choices determined by the environment

In the simple case of two Petri net fragments each having only one initial place and one final place, the **alt** combinator constructs a Petri net fragment containing copies of both fragments with these places shared between them. An example is shown in Figure 3.12. The Petri net P is equivalent to **seq** (**get** a , **put** b) after routine local optimizations (cf. Figure 3.10), and Q is equivalent to **seq** (**get** c , **put** d). Because the token in the shared initial place can go only one way or the other, the combination $\text{alt}(P, Q)$ either acknowledges an input of a with an output of b , or acknowledges

an input of c with an output of d , but not both. If the environment were to send both inputs, only one would be acknowledged, with the choice being made non-deterministically by the circuit if necessary. However, in most designs including this one, the intended effect is for the flow of control through the Petri net to be directed by signals received from an appropriate environment.

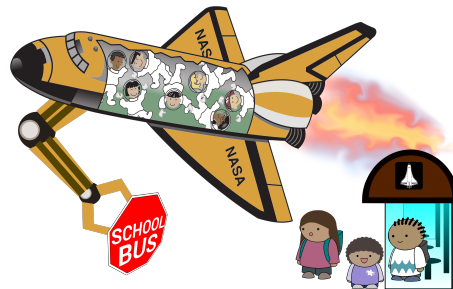
Provision for atomicity

In the general case, Petri net fragments may have multiple initial and final places, as any Petri net fragment resulting from the **par** combinator typically does. This case is more complicated for the **alt** combinator because of some extra housekeeping needed to ensure mutual exclusion with atomicity (*i.e.*, “all or nothing” selection of one alternative or the other). Provision for the general case is part of the formal specification of the **alt** combinator in [Section 5.4.5](#), which this informal discussion omits. As a result, the necessary invariants are maintained automatically whenever the **alt** combinator is invoked, without any extra effort required of the designer.

3.6.4 Adaptation to an environment

While it is always important to meet the demand for circuits of ever increasing sophistication, there can also be cost benefits in making them no more capable than necessary for their assigned job, and sometimes even performance benefits as well if doing so makes them smaller and faster. For example, a common practice in conventional logic design is to exploit any input conditions expected not to occur in deployment (so called “don’t care” conditions) by designing the circuit to respond to them in whatever way simplifies the implementation [38, 53].

In a similar vein, it is sometimes convenient to overspecify DI circuits at first and then to relieve them of unnecessary obligations afterwards. A natural way of proceeding is to describe the expected environment for a circuit using process combinators just as we would describe the circuit, except that the inputs to the circuit are the outputs from the environment and *vice versa*. The effect of using the **env** combinator to combine the circuit with the environment is a specification similar to the original circuit but incapable of any behavior not exercised by interacting with the given environment.



When to use the env combinator

Even though this approach seems to require writing two specifications instead of one, it is often a net win. The environment specification is typically much simpler than the circuit, comparable to a test driver in software development. Moreover, in some realistic applications, especially those involving circuits that interact with multiple non-interacting peers, attempting to specify exactly the minimum required behavior of a circuit by itself is considerably more difficult than achieving the same effect by approximating it in conjunction with an environment. The classic example is the so called *nacking arbiter* [133], which is something like a non-blocking version of the sequencer described in [Section 2.4](#).

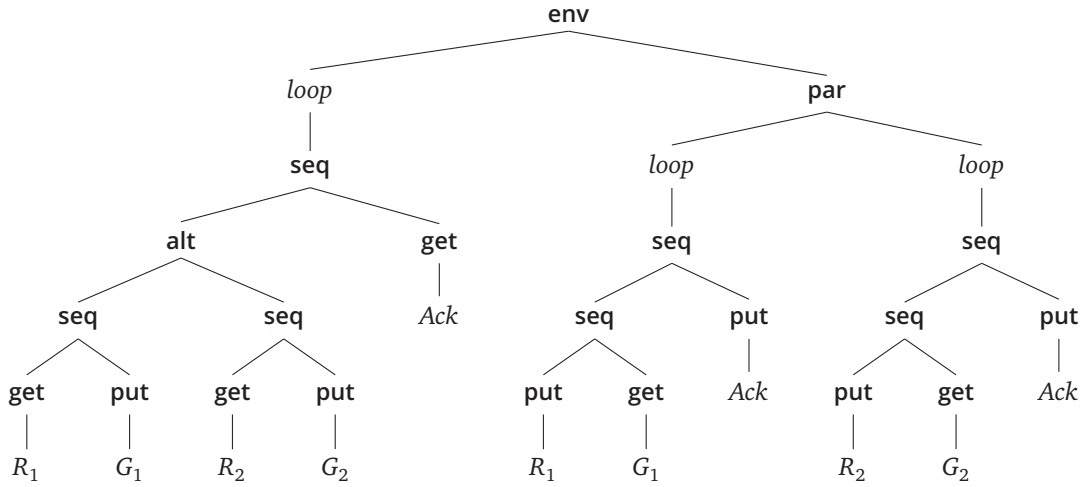


Figure 3.13: abstract syntax of the procedural description of a 2-way sequencer

Example usage of the env combinator

A simpler example than the nacking arbiter is a 2-way version of the sequencer as given by the specification shown in Figure 3.13, which is an abstract syntax tree such as a compiler might use to represent the following process combinator expression. This expression also incidentally exemplifies the full set of combinators including `fix` by way of Equation 3.5.

$$\begin{aligned}
 S = \text{env} (& \hspace{15em} (3.6) \\
 & \text{loop seq (alt (seq (get } R_1, \text{put } G_1), \text{seq (get } R_2, \text{put } G_2)), \text{get Ack}),} \\
 & \text{par (loop seq (seq (put } R_1, \text{get } G_1), \text{put Ack}), \text{loop seq (seq (put } R_2, \text{get } G_2), \text{put Ack}))}
 \end{aligned}$$

In this example, the environment is a pair of concurrent processes, each repeatedly and independently issuing requests and acknowledgments to the sequencer. Although it may not be obvious, there is a difference between the sequencer alone

$$T = \text{loop seq (alt (seq (get } R_1, \text{put } G_1), \text{seq (get } R_2, \text{put } G_2)), \text{get Ack})} \quad (3.7)$$

and the combined system S . Due to the Petri net semantics, the sequencer specification T by itself without the environment would mean an acknowledgment arriving in advance of an initial request should be buffered until needed. Although it would do no harm, this capability is not used in practice, so the environment specification revokes it, thereby allowing for a less costly implementation.

Implementation of the env combinator

Despite its expressive power, the implementation of the `env` combinator is strikingly simple, at least in the special case where it occurs at the outermost or root level of a process combinator expression. As shown in Figure 3.14, it is similar to the `par` combinator except that it does not anonymize any shared transitions (cf. Figure 3.11). By forming a closed system containing the circuit specification and its environment, this operation also drastically reduces the space of reachable markings in

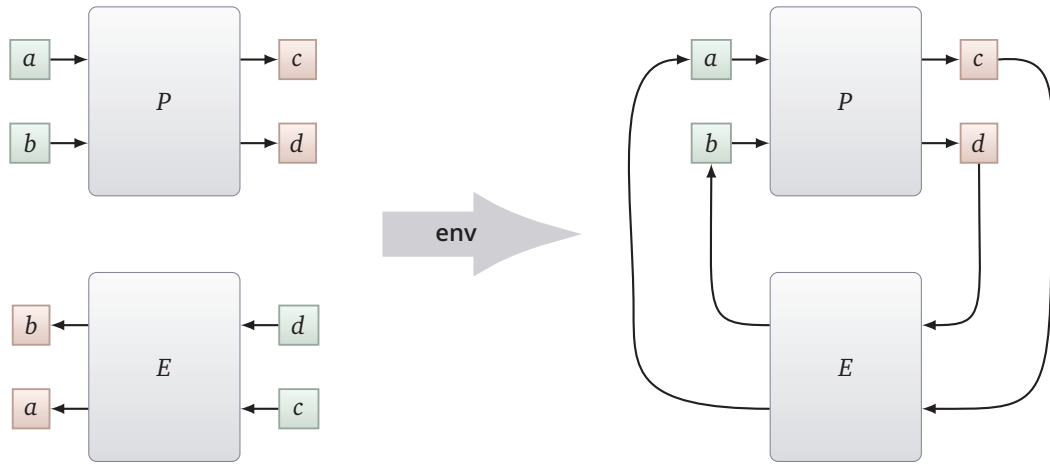


Figure 3.14: For Petri net fragments P and E , the combination $\text{env}(P, E)$ behaves like P only as far as necessary to interact with E .

practice, which has certain computational benefits. In the general case, an additional semantics-preserving transformation described in [Chapter 7](#) is automatically invoked along with the env combinator to maintain the invariant of open input and output transitions, as required when the result is used as an operand to other combinators.

Brainteasers

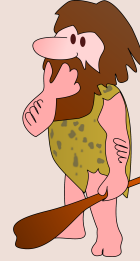
1. A mathematically minded colleague seeks to discover the algebraic laws governing process combinators. He proposes a “do nothing” process given by

$$\text{skip} = \text{seq}(\text{put } x, \text{get } x)$$

as an identity element for the **seq** and **par** combinators, in that the following behavioral equivalences should hold for all processes P .

$$P \equiv \text{seq}(\text{skip}, P) \equiv \text{seq}(P, \text{skip})$$

$$P \equiv \text{par}(\text{skip}, P) \equiv \text{par}(P, \text{skip})$$



- a) Sketch the Petri net model of the *skip* process. What happens to the x ? (hint: See [Figure 3.11](#).)
- b) What would be another way to define a “do nothing” process?
- c) Define a process like *skip*, but having the algebraic properties of an identity element for the **alt** combinator. What would be a good name for it?
- d) Which of these equivalences could fail in some contexts? Demonstrate by constructing a process combinator expression that can behave differently when the substitution is made. (hint: It involves the **alt** combinator and may call for a sketch or two. See the next question for inspiration.) What explains this apparent loss of compositionality? (See [page 53](#) for a definition.)
2. A contract programmer tasked with implementing a process combinator library proposes simplifying the specification by requiring all Petri net fragments to have exactly one initial place and exactly one final place. This condition is easily met by redefining the **par** combinator as shown in [Figure 3.15](#), and has the purported benefits of making the **alt** combinator implementation always resemble the simple case of [Figure 3.12](#) and beautifying the algebraic laws.

- a) Sketch the Petri net model of the process P using the proposed **par** combinator definition.

$$P = \text{alt}(\text{seq}(\text{get } a, \text{put } x), \text{seq}(\text{par}(\text{get } b, \text{get } c), \text{put } y))$$

- b) Based on the expression, what should happen if the environment sends a to P ?
- c) Based on the sketch, what might really happen instead?
- d) Does the contractor have a good idea?
3. Construct an exact behavioral equivalent to [Equation 3.6](#) without using the **env** combinator. (hint: [Equation 16.6](#)) What could be done in principle to convince a skeptical reviewer of the equivalence?

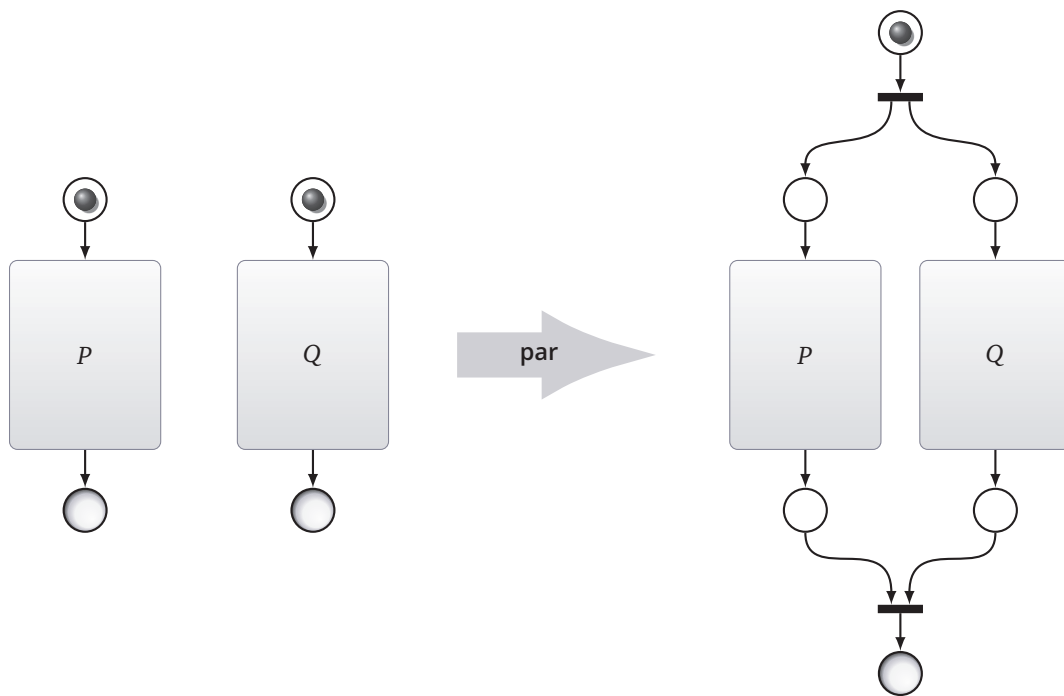


Figure 3.15: An alternative definition of the **par** combinator maintains a single initial and final place as an invariant for all Petri net fragments (cf. Figure 3.11).

Think not those faithful who praise
all thy words and actions, but those
who kindly reprove thy faults.

Socrates

CHAPTER  4

SUCCESS

By opting for a Petri net model over a naive trace structural semantics, we gain a useful *repertoire* of computationally efficient composition operations, interoperable translation targets for textual and graphical circuit description formalisms, and a satisfactory treatment of progress obligations. It now remains to incorporate analytical features into this semantic framework so that questions bearing on correctness can be settled, thereby addressing the last remaining point noted in [Section 3.4.2](#). For this purpose, we resume a line of investigation toward a workable extensional process model involving trace analysis.



4.1 Reachability graphs

A first step in this direction is the transformation of a Petri net to its verbose relative known as a **reachability graph**. The reachability graph is a graph in the formal sense of a collection of vertices connected by directed edges. Each vertex in the reachability graph corresponds not to a place or transition in the Petri net, but to an instance of the whole Petri net with a particular marking (*i.e.*, a particular assignment of tokens to places). An edge from one vertex in the reachability graph to another indicates that the marking associated with the latter is obtained from that of the former by the firing of an enabled transition. Not every possible assignment of tokens to places is necessarily a reachable marking. Only the initial marking and those obtained directly or indirectly from it by firing enabled transitions are included in the graph, hence the terminology.

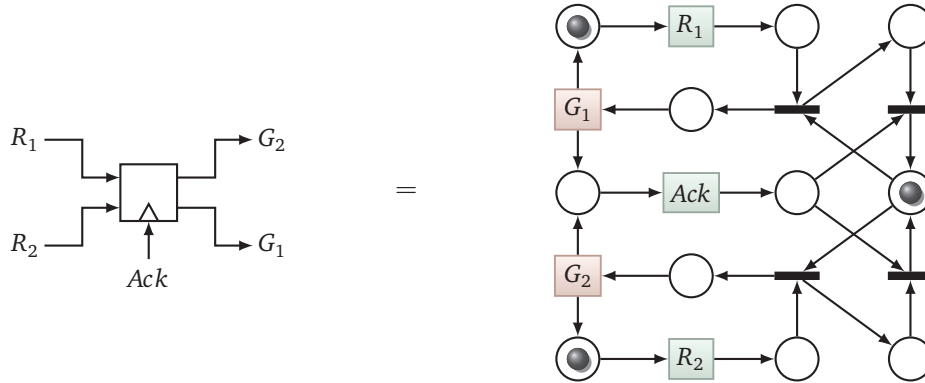


Figure 4.1: Petri net model of a 2-way sequencer in an environment

4.1.1 Example of a reachability graph

For concreteness, we can make further use of the sequencer example from Section 3.6.4. Shown in Figure 4.1 is a Petri net model resulting from Equation 3.6 with its initial marking, after some aggressive optimization using Petri net reductions [158, 219, 253]. In the initial marking, the transitions R_1 and R_2 are enabled. Firing either of them results in a new marking. Each of these new markings is therefore connected to the initial marking by an edge in the reachability graph. We may infer that the reachability graph contains at least the three vertices in Figure 4.2 connected as shown. However, the graph does not end here, because each of the new markings contains two enabled transitions. Firing them leads to other markings that need to be added to the graph, and these lead to further markings. Continuing until no further markings are found leads to the graph in Figure 4.3.

4.1.2 Reachability graph algorithms

Building a reachability graph is costly but straightforward in principle by generalizing from this example to Algorithm 4.1. An algorithm like this one might not terminate if unsafe markings are explored (see page 53) because in that case arbitrarily many tokens could accumulate in the same place, with each multiplicity constituting a distinct marking. However, termination is guaranteed by refraining from enumerating the successors of unsafe markings, which contribute no further information about the behavior of the system being modeled. (This issue does not affect the current example, wherein all reachable markings are safe.)

Because reachability graph sizes tend to increase exponentially with those of the Petri nets associated with them, more sophisticated algorithms than this one are needed in practice to cope with them effectively. One well known family of algorithms is based on the *stubborn set* method [241, 290, 291, 292] another, the *sleep set* method [98], and another, the *ample set* method [227], all of which can be classified as *partial order reduction* methods [59]. These aim to save time by generating only a subset of the reachable markings while trying not to miss those deemed relevant to the property under consideration (e.g., safety, liveness, or in our case, language preservation). Orthogonal to these efforts, data structures suitable for compact storage and fast retrieval of large

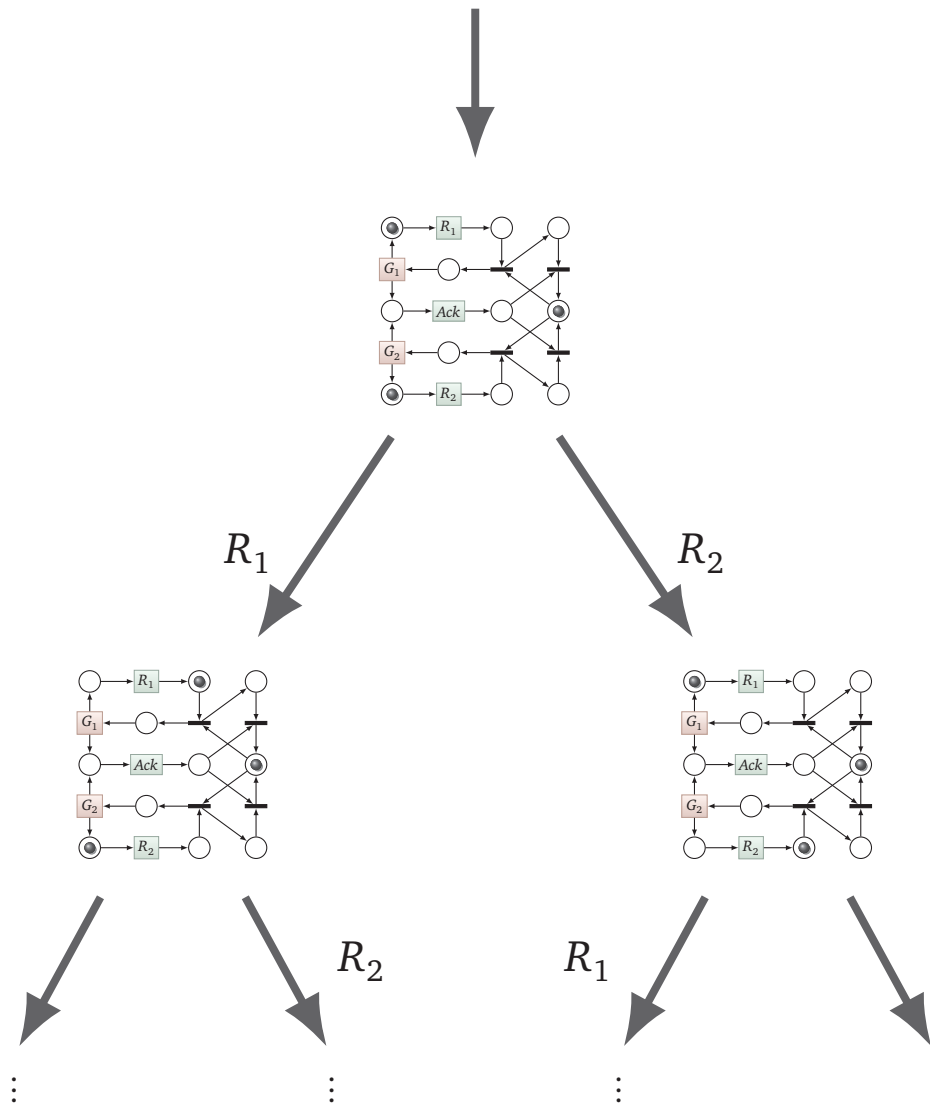


Figure 4.2: The reachability graph of the sequencer in Figure 4.1 is enumerated starting from the initial marking by searching for enabled transitions and firing them. Each vertex represents a distinct marking.

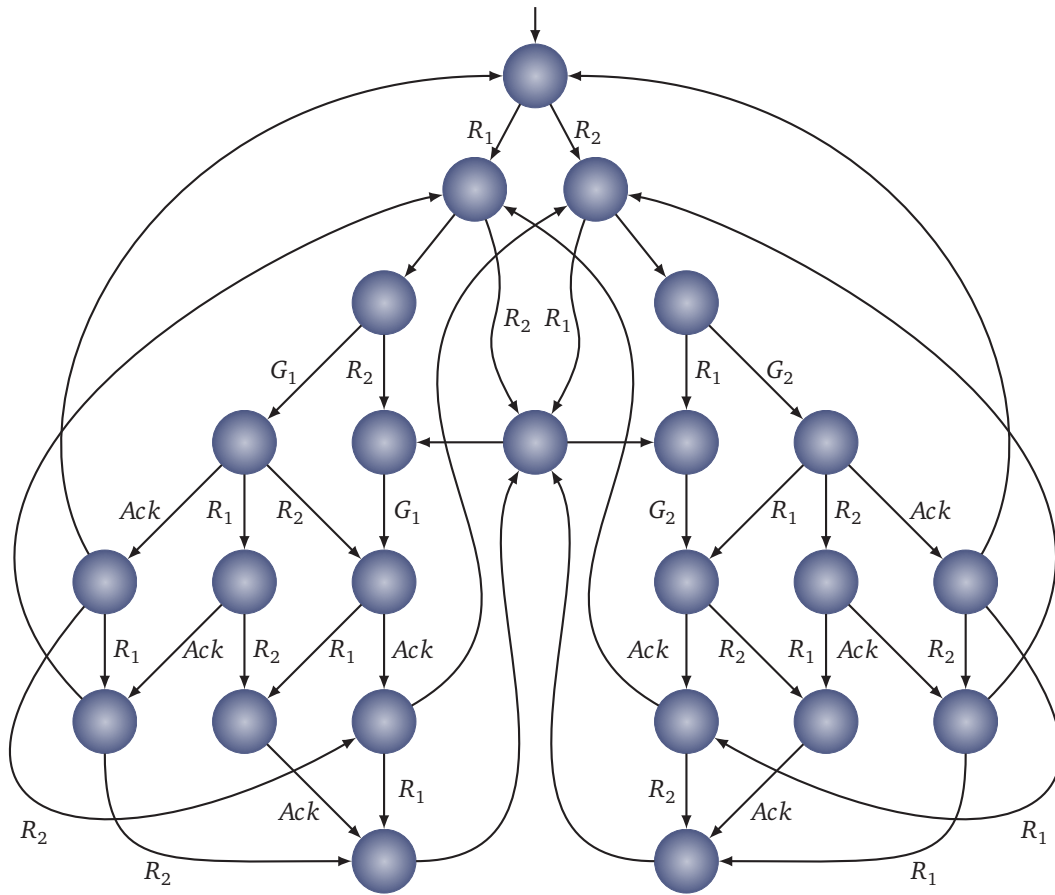


Figure 4.3: Extending Figure 4.2 as far as possible leads to the full reachability graph. Edges due to named transitions are labeled accordingly.

sets of markings allow for the solution of larger problems than would be feasible otherwise (in many cases). These include **binary decision diagrams** [41] and related methods involving hierarchical decomposition [45, 56].

Further optimizations are possible if we decouple the reachability graph from the Petri net model by relaxing the requirement of a precise correspondence between vertices and markings. This flexibility permits another small step toward extensionality by upgrading the reachability graph to an independent description of the process, amenable as such to comparable notions of semantics-preserving transformation. A straightforward adaptation of the standard textbook algorithm for state machine minimization by partitioning [115], whereby behaviorally equivalent states are merged, does the same for a reachability graph. This optimization entails the merger of all unsafe markings, and all markings connected to an unsafe marking by an output or an unlabeled transition, which for practical purposes are also unsafe. Safe markings reachable as a result only through unsafe markings may be deleted. Furthermore, most unlabeled edges can be removed according to a simple


```

Input: An initially marked Petri net
Output: A reachability graph

1: Create the graph of a single vertex with the initial marking.
2: repeat
3:   Select an unvisited vertex  $N$  with marking  $M$  from the graph.
4:   if  $M$  is safe then
5:     for each transition  $T$  enabled in  $M$  do
6:       Generate a marking  $M'$  from  $M$  by firing  $T$ .
7:       if no vertex with marking  $M'$  is already in the graph then
8:         Add a vertex with marking  $M'$  to the graph.
9:         Add an edge to the graph from  $N$  to the vertex with marking  $M'$ .
10:    Mark  $N$  visited.
11: until there are no more unvisited vertices in the graph

```

Algorithm 4.1: one way to build a reachability graph

rewrite rule without altering the observable firing sequences encoded by the graph. The effect of these additional optimizations on the current example is shown in [Figure 4.4](#).

4.2 The transducer model

Even in its most compact form, the reachability graph leaves something to be desired as an aid to intuition or as an object of analysis. Its real purpose is to be an intermediate representation bridging the gap between a Petri net and a transducer model (*cf.* [Figure 3.1](#)). A transducer model can be made more compact than the corresponding reachability graph by labeling its edges with sets of concurrent signals if their order is immaterial. Significant portions of the reachability graph concerned with expressing interleaving among signals are unnecessary in the transducer, leading to a graph with fewer vertices as shown in [Figure 4.5](#), yet another step removed from Petri net markings.

The algorithm to construct a transducer starts with an initial version isomorphic to the optimized reachability graph excluding unsafe markings, which are not explicitly represented in the transducer.¹ In the initial version, each edge is labeled with at most one input or output signal, as in the reachability graph. The transducer is then simplified according to semantics-preserving local reduction rules, resulting in a form that generally entails multiple edge labels. A final minimization phase removes unreachable or redundant vertices by partitioning.

4.2.1 Operation

The intuitive interpretation of a transducer as a process is as follows.

¹An empty transducer results from the degenerate case of the reachability graph containing only unsafe markings.

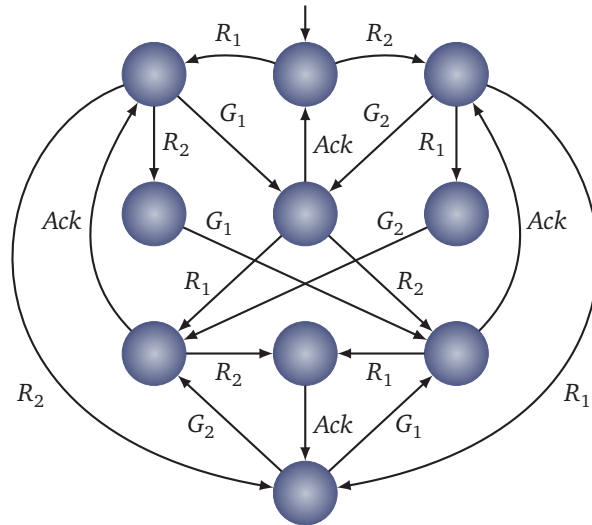


Figure 4.4: A reduced reachability graph obtained from Figure 4.3 by local optimizations eliminates unlabeled edges and many vertices.

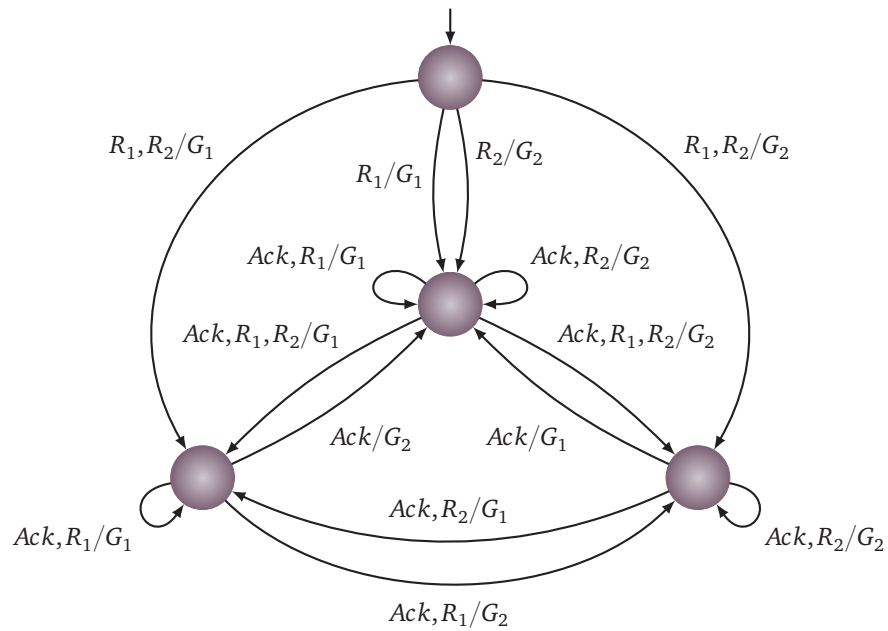
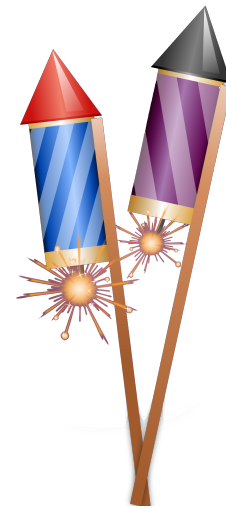


Figure 4.5: Edges labeled with i/o bursts enable an even more compact description than the reduced reachability graph as a transducer model, with vertices no longer corresponding to markings.

- Each vertex represents a state of the process, of which the process may occupy only one at a time. An initial state is distinguished in the diagram by an unlabeled incoming edge with no origin.
- For clarity, the set of signals labeling each edge is partitioned between a set of inputs (the *input burst*) and a set of outputs (the *output burst*).
- Upon finding the process in a particular state, the environment may transmit any input burst associated with an outgoing edge from that state.
- Upon receiving an acceptable input burst in its entirety, the process assumes the state reached by traversing the associated edge, while simultaneously transmitting the corresponding output burst to the environment.

Unlike the situation of a burst mode automaton (page 36), input bursts and output bursts may happen concurrently within these constraints. That is, the environment may issue an input from the moment it makes enough of a determination of the current state to establish its acceptability, even if an output burst is still in progress or not fully received. If there were more than one possibility, this determination would be based on the receipt of a disambiguating subset of the output burst associated with a transition from the previous state.

It should also be noted in passing that there is no prohibition on input bursts being proper subsets of one another, or of identical input bursts leading to different states. The process may make non-deterministic choices in these cases.



4.2.2 Limitations

Being arguably clearer and more intuitive than the Petri net or reachability graph, the transducer model may be used to establish confidence in a design by visual inspection or manual simulation alone in simple cases. In light of this possibility, an inquisitive reader might wonder why the transducer is not advocated as a method of human-writable specification along with block diagrams and procedural descriptions via process combinators. The answer is that constructing transducers manually would probably be a bad idea for reasons of delay insensitivity, compositionality, and abstraction.

Delay insensitivity Not every transducer graph that can be drafted manually makes sense as a model of a delay insensitive process. For example, an output enabled by an input burst must not be precluded in the same state by a proper superset of that input burst [288]. Conditions like these should not be the responsibility of a designer to establish manually, but they are always met when the transducer is automatically generated.

Compositionality The transducer model of a network of interacting transducers increases in size as the product of their state spaces, bears no resemblance to any of them, and is no mean feat to construct either manually or automatically. The transducer contrasts sharply with the Petri net

model in this regard (Section 3.5.3), making the latter essential as an intermediate representation in any non-trivial design.

Abstraction Transducers lose their intuitive appeal as they grow to moderate sizes, which happens easily. An n -bit memory register is modeled by a transducer with 2^n states and an even more rapidly growing number of edges. When a family of circuits is parameterized by a natural number n , a better approach than solving each case separately is to strive for a general form. A domain specific language based on process combinators with adequate abstraction features is better equipped for this challenge than an *ad hoc* practice of transducer construction. For spatially iterated structures such as registers, even block diagrams are preferable.

4.2.3 Utility

The transducer model is nevertheless a helpful accessory to the design work flow for its use in Petri net optimization, state based synthesis, and verification as explained below.

Petri net optimization One interesting characteristic of the transducer model is its ability to be transformed automatically back to an equivalent Petri net model often simpler than the one from which it is derived. Where feasible, the round trip from an initial Petri net through a reachability graph, through a transducer, and back to a Petri net would make an excellent global optimization algorithm if the most compact attainable representation of a process as a Petri net were of interest for any reason. A version of this algorithm is actually used in connection with the `env` combinator (Section 3.6.4) when it is necessary to transform a closed Petri net to one with open input and output transitions.



State-based synthesis The transducer can also be the input source for a class of state-based circuit synthesis algorithms targeting the flat netlist representation. A standard construction consists of a sequencer as a front end serving to serialize the inputs to a network encoding a state table, which maps inputs and current states to outputs and successor states. Wires act effectively as state-holding elements [221]. Generalizations and enhancements to this basic construction are reviewed extensively in Chapter 15 because they form the theoretical foundation for a crucial link in an automated development tool chain.

Regarding the development work flow, an important implication of this class of algorithms is the completion of a course through Figure 3.1 starting from the procedural description and leading to the flat netlist by way of the reachability graph and the transducer. Given a feasible technology mapping path from the flat netlist representation, this methodology enables the design of space efficient DI circuits from end to end with minimal hardware knowledge required of the novice designer, and frees the expert designer to focus on the aspects that can not be automated. This route is also a slight improvement over the block diagram to flat netlist work flow mentioned in Section 3.3.3 because it allows a better chance of detecting obvious errors by inspection of the transducer. The space efficiency of circuits synthesized by this method will normally surpass the results of alternative algorithms for direct mapping synthesis from Petri nets to netlists (Chapter 16) unless the circuit is of some particularly repetitious form. However, this benefit comes at the cost of state space enumeration as a prerequisite to synthesis, which can be prohibitive for large systems.

Verification Equally as important as synthesis is the use of the transducer model for verification. Although there is no designated canonical form, behaviorally equivalent designs and specifications tend to converge to identical transducer models up to isomorphism, facilitating comparison often without need of further analysis. As noted at the beginning of [Section 4.2.2](#), a subjective verification can suffice in simple cases. For more complicated cases, the transducer gives rise to an ensemble of finite automata describing the behavior of the process in a way that lends itself to efficiently computable equivalence and refinement relations. These models are described in [Section 4.4](#) following a brief theoretical primer.

4.3 Traces revisited

Parsimonious treatments of behavioral equivalence and refinement win by ignoring inessential differences between processes. Whatever its flaws, the trace semantics contemplated in [Section 3.4](#), by virtue of omitting all operational details, approaches this ideal of extensionality more closely than the alternatives considered thereafter. On the other hand, having met the needs for prescriptive and descriptive semantic models via the latter, we can focus more effectively on the analytical at this point by reconsidering a trace semantics without being obliged to account for process composition in terms of trace theory.

4.3.1 Progress obligations

As noted previously, one problem with a trace set is its inability to express progress obligations. What a process is allowed to do and what it is obligated to do by its specification can be two different things. For example, the complete specification of a process with an input a , an output b , and a trace set $\{\epsilon, a, ab, aba, abab, \dots\}$ would need an additional bit of information about each trace to say whether the process is required to output upon completing it. Perhaps the designer has it in mind that by engaging in aba , the process enters into a binding commitment to output another b . Maybe by the time the process completes $ababa$ it is free either to halt or to continue. The progress obligation in the first case can not be inferred merely from the presence of longer traces in the set. As the theory stands, any extra conditions like these would need to be stipulated in some *ad hoc* way.

4.3.2 Quiescent traces

A more systematic way of preserving these distinctions would be to record an additional set containing only the traces that are definitely required to be followed up by further output. In the current example, aba would be a member of this extra set, but $ababa$ would not, so the distinction would be clear. Process models using more than a single trace set are not unprecedented (e.g., [\[55, 72\]](#)), and many variations are possible. An opposite but equally valid convention would be to note the set of traces whereupon the process may refuse to output (or definitely will refuse) instead of being required to output. These traces are usually called *quiescent*.²

In addition, the latter convention allows a summary of the behavior of a process including progress obligations with a single set, because the quiescent trace set requires no other information about the process to recover the original trace set. Forming the set of all prefixes of quiescent traces (the so called *prefix closure* of the quiescent trace set) would recover it if required for any reason.

²but see [\[243\]](#) for a testament to the disarray of terminology in this area

For example, the prefix closure of a quiescent trace set $\{\epsilon, ab, abab, ababab, \dots\}$ would include the rest of the traces such as $a, aba, ababa$, and so on.

4.3.3 Refinement

Thanks to the effective treatment of progress obligations, the quiescent trace set would seem sufficiently expressive to enable well informed determinations of compatibility between processes. It should be justifiable at least to assert the behavioral equivalence of any two processes whose quiescent trace sets are identical.³ However, behavioral equivalence is usually of secondary interest to the more practical propositions of whether a circuit meets its specification, or whether one circuit is a compatible replacement for another. Although it implies these conditions, demanding behavioral equivalence in cases like these places the bar higher than necessary and is often inconvenient or inappropriate. An already well behaved circuit should not require modification for the sake of an ill-conceived theoretical property.



The weaker condition we seek in these situations goes by the name of **refinement**, as noted previously. If a process T is a compatible replacement for a process S in any environment, then T is called a refinement of S , or is said to **refine** S . A circuit T correctly implementing a specification S need not be equivalent to S , but must refine it. If T refines S but is not equivalent to S , we envision T as having extra features not possessed by S but not in conflict with it. Refinement therefore is not a symmetric relation, but a pair of processes may nevertheless refine each other if and only if they happen to be equivalent. For the mathematically minded reader trying to keep track, it follows also that refinement is transitive and reflexive, hence a partial order relation.

4.3.4 Trace analysis

If the refinement relation is computable at all and if a quiescent trace set completely determines a process, intuition suggests the existence of a decision procedure for refinement taking as input only a pair of quiescent trace sets (in some finite concrete representation). In fact, quiescent trace sets arising in practical applications belong to the class of regular languages, making most standard decision problems about them computable by well known algorithms [115]. The difficulty lies only in formulating the exact criteria corresponding to refinement as properties of the sets.

Quiescent trace set containment

An obvious suggestion is to identify refinement with quiescent trace set containment based on the intuition that a process T refining a process S can do anything S can do and maybe more. However, this proposal is destined to be short-lived. If S has quiescent traces $\{\epsilon, ab, abab, ababab, \dots\}$ and T has all those in addition to aba , then the containment holds but the refinement does not. S must always acknowledge a with b , but T has the potential to stop upon the second receipt of a , thereby shirking a progress obligation of S . Relying on T as a replacement for S therefore could deadlock the system.

³ignoring fairness, real-time deadlines, fault tolerance, or other criteria beyond the scope of trace models

Divergent trace sets

A more nuanced intuition would entail the idea that T may differ freely in good ways from the process S it refines, but must refrain from any undesirable innovations, whatever that might mean. Taking this notion seriously if possible calls for clearer conventions regarding undesirable behavior.

The complement of the set of acceptable traces is the set of traces representing breaches of protocol between a process and its environment. Some of these prohibited traces represent misbehavior of the process, and the rest that of the environment. The latter are called the **divergences** of the process, whereas the former have no conventional designation. To be a divergence, a trace necessarily begins with some (possibly empty) prefix of an acceptable trace, and then goes wrong due to an unauthorized input signal (sent by the environment), with any signals occurring subsequently in it unconstrained. With the possible exceptions of some degenerate cases, any process has an infinite set of divergences, because anything at all may follow after the first bad input in a divergence with no prospect of redemption.

Combined quiescent and divergent trace set containment

The key insight attributable to [128] is that a useful concept of refinement is indeed captured by a set containment relation, albeit one of a subtle form. Following the seemingly whimsical step of mixing a process T 's set of quiescent traces Q_T with its divergences D_T into a combined set $R_T = Q_T \cup D_T$, and similarly obtaining $R_S = Q_S \cup D_S$ for a process S , the astonishing outcome is that the formal criterion $R_T \subseteq R_S$ coincides almost invariably with the intuitive notion of T refining S pursued above. This relation therefore is taken as the definition of refinement, which mathematically minded readers will note is a partial order as expected. However, the definition is contingent on the processes in question having identical alphabets, without which refinement is impossible (and irrelevant) due to obviously incompatible interfaces.

Terminology in this context has a troubled history, with divergent and quiescent traces sometimes designated collectively as **failure traces** or just plain failures in the literature. Cognate internet memes aside, some authors not unreasonably reserve the word “failure” for traces pertaining to something actually failing [72], and others appear to have used it synonymously with quiescence [128]. The lack of any generally accepted intuitively transparent term prompts the neutral designation of **refinement relational traces** for members of the set $R = Q \cup D$ hereafter in this book, or relational traces more briefly.



4.4 From transducers to trace recognizers

To compute the refinement relation in practice requires a concrete representation of the relational trace set. A conventional textbook-style state machine is suitable for this purpose. Also known as a **deterministic finite automaton** (DFA), it can be described informally as a graph of states joined by labeled directed edges, with a distinguished initial state and a subset of the states designated as **accepting** states. A DFA is said to **recognize** the set of all sequences of symbols in which each member sequence is spelled out by the edge labels along a walk from the initial state to an accepting state. The set of sequences recognized by a DFA is also called its **language**. As the name implies,

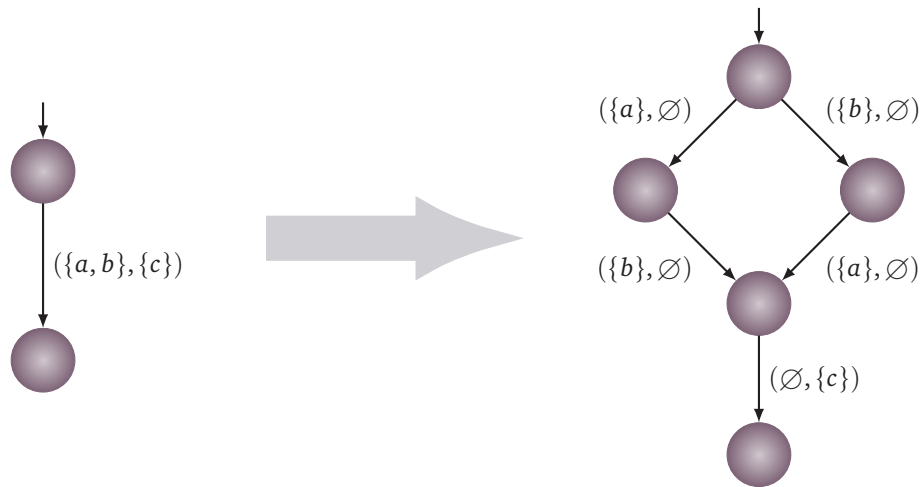


Figure 4.6: In preparation to derive the relational trace set recognizer, i/o bursts in a transducer model are transformed to singleton sets by introducing intermediate states.

a DFA is always finite even if the language it recognizes is infinite, which is possible if the graph contains cycles (details on page 160).

This theory is of immediate interest for two reasons. One is that a transducer model such as the one shown in Figure 4.5 can be transformed automatically into a corresponding relational trace set recognizer by a method to be outlined presently. The other is that well understood decision procedures (whose details need not concern us at the moment) exist for testing containment between any two languages given by their recognizers [1, 115]. We therefore have the means to test refinement automatically between circuit designs or specifications originating from block diagrams or process combinators by transforming them to the relational trace set representation and applying this decision procedure. This capability enables a regime of quality control whereby bugs introduced through modifications to a previously verified design need never escape detection.

4.4.1 A preliminary subgraph

The derivation of a relational trace set recognizer from a transducer proceeds first by rewriting the transducer to limit every i/o burst to at most one input or output signal. As shown in Figure 4.6, this transformation is always possible by introducing new intermediate states where needed.⁴ The graph thus obtained is isomorphic to a subgraph of the relational trace set recognizer, with the initial state of the transducer mapping to the initial state of the relational trace set recognizer, and the edges similarly labeled as shown in Figure 4.7.

The accepting states in this subgraph are determined by the quiescent traces of the process it represents, which the relational trace recognizer must accept. A quiescent trace by definition leads

⁴This step does not necessarily restore the transducer to an isomorphism of its antecedent reachability graph, because intervening transducer-specific optimizations may have applied.

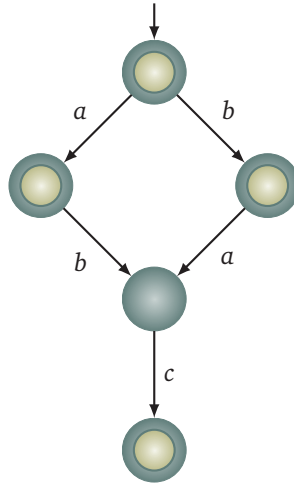


Figure 4.7: A subgraph of the relational trace set recognizer is isomorphic to the expanded transducer. Accepting states depicted with an inner circle are those without an output-labeled outgoing edge.

to a state from which no further progress is made until the environment supplies an input. A state having a way forward through an output-labeled edge is not quiescent, and hence not accepting. There is one such state in Figure 4.7, implying that all other states are accepting.

4.4.2 The complete graph

This construction is incomplete as a DFA because for many possible traces it gives no indication of whether or not they are accepted. In a well defined DFA, every state needs exactly one outgoing edge labeled by each member of the alphabet. Adding the missing edges to the graph also requires adding the right states for them to reach. Fortunately, in any relational trace set recognizer, only two additional states are ever needed. Each of the additional states is a **trap state**, meaning that every outgoing edge points back to it. One of the trap states is designated as an accepting state, and the other is not.

A simple rule supplies the remaining edges needed in the graph. For each state, we create any missing output-labeled edges and connect them to the non-accepting trap state. Any missing input-labeled edges are created and connected to the accepting trap state. These latter edges correspond to the first prohibited input occurring by definition in divergent traces, which the relational trace recognizer must accept. The resulting graph for the current example is shown in Figure 4.8.

4.4.3 Edge cases

One technical point not evident from this illustration is that a state in the preliminary subgraph could have too many outgoing edges rather than not enough, with some edges having the same label as others. This situation occurs in cases of non-deterministic behavior reflected in the transducer model. A similar procedure for building the full graph applies, resulting in a **non-deterministic**

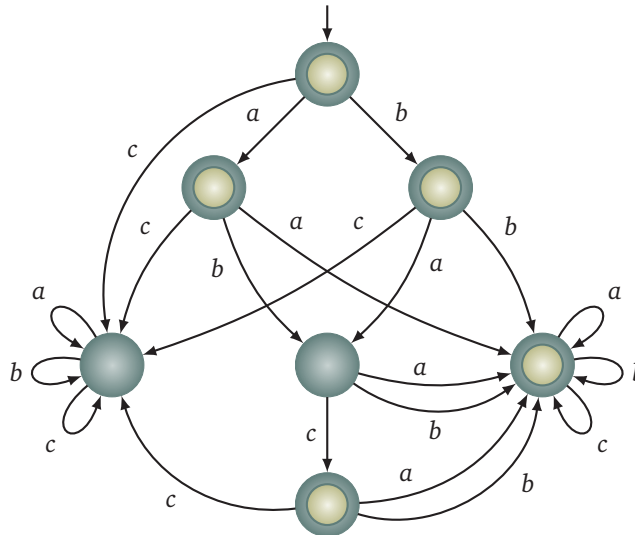


Figure 4.8: Two trap states and a full complement of edges finish the job of constructing a relational trace recognizer.

finite automaton (NFA). The NFA can be converted to an equivalent DFA via the standard power set construction [115], or can be used for refinement testing directly by efficient algorithms discovered more recently [1].

Another technicality pertains to the case of an empty transducer resulting from a reachability graph with an unsafe initial marking. In this case, the preliminary subgraph is also empty, but the trap states are to be added to it nevertheless. The only adjustment needed to accommodate this situation is to take the accepting trap state as the initial state. This convention makes every trace a relational trace, so that the relational trace set is too large to be contained in any other trace set. Under the refinement ordering, such a process does not meet any specification but itself.

4.4.4 Other trace recognizers

As illustrated in Figure 3.1, the relational trace recognizer can be transformed automatically to a quiescent trace recognizer and to a divergent trace recognizer. These transformations are trivial. The quiescent trace recognizer is obtained by changing the accepting trap state to non-accepting, and the divergence recognizer by making the accepting trap state the only accepting state.

Usage

These other automata are not needed for checking refinement, but may help in understanding or troubleshooting a design by determining a set of traces attesting to discrepancies between processes, which can be more informative than the knowledge of their refinement ordering alone. For example, subtracting the quiescent trace set of an intended specification from that of its proposed implementation can reveal the specific chain of events leading to a failure due to deadlock. Similarly,

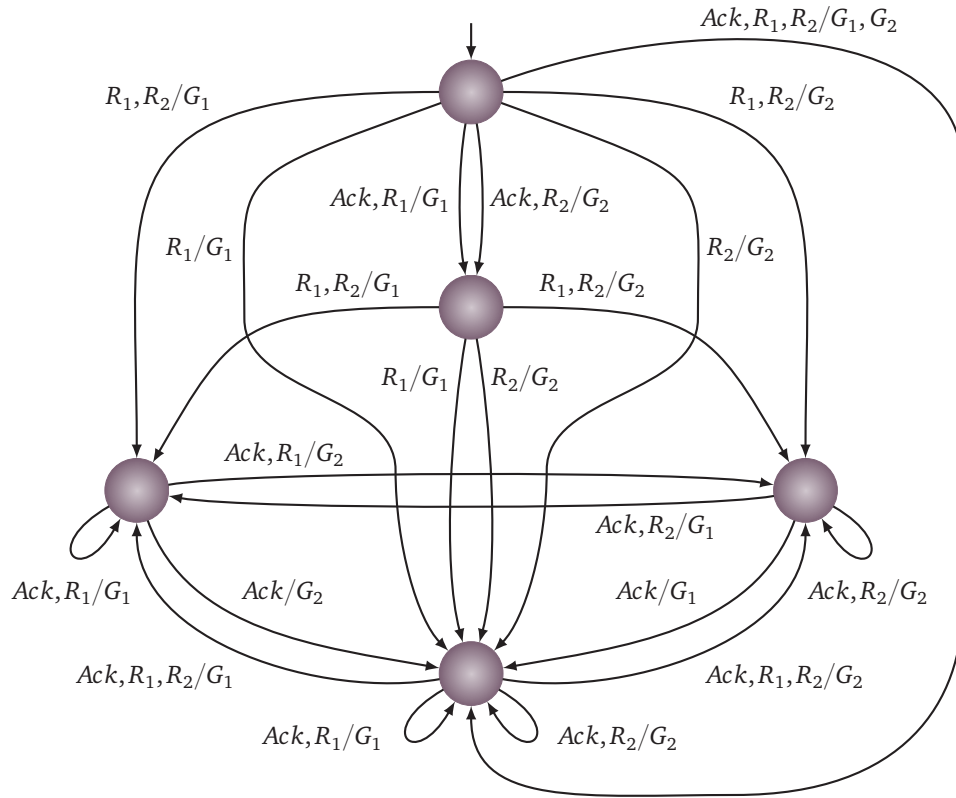


Figure 4.9: transducer model of the process defined in Equation 3.7, which is a refinement of the 2-way sequencer as defined in Equation 3.6, whose transducer model is shown in Figure 4.5

the difference between the divergent trace sets can indicate the combination of inputs that break an incorrect implementation. These operations, along with any based on union or intersection, can be performed efficiently on sets represented as regular automata.

A familiar example

A concrete example of the sort of inquiry facilitated specifically by quiescent and divergent trace recognizers comes from an aspect of the sequencer design mentioned briefly in Section 3.6.4. The claim is that the process T described by Equation 3.7 is more complex and does more than necessary for the sequencer specification S described in Equation 3.6, albeit compatible with it (hence the use case for the `env` combinator). Visual comparison of the transducer models for S and T in Figure 4.5 and Figure 4.9 respectively shows the latter to be more complex. The refinement relation, though not immediately evident from the transducer models, is confirmed as expected by algorithmic comparison of their relational trace sets.

The complexity of T relative to S may seem counterintuitive in view of the simpler form of its

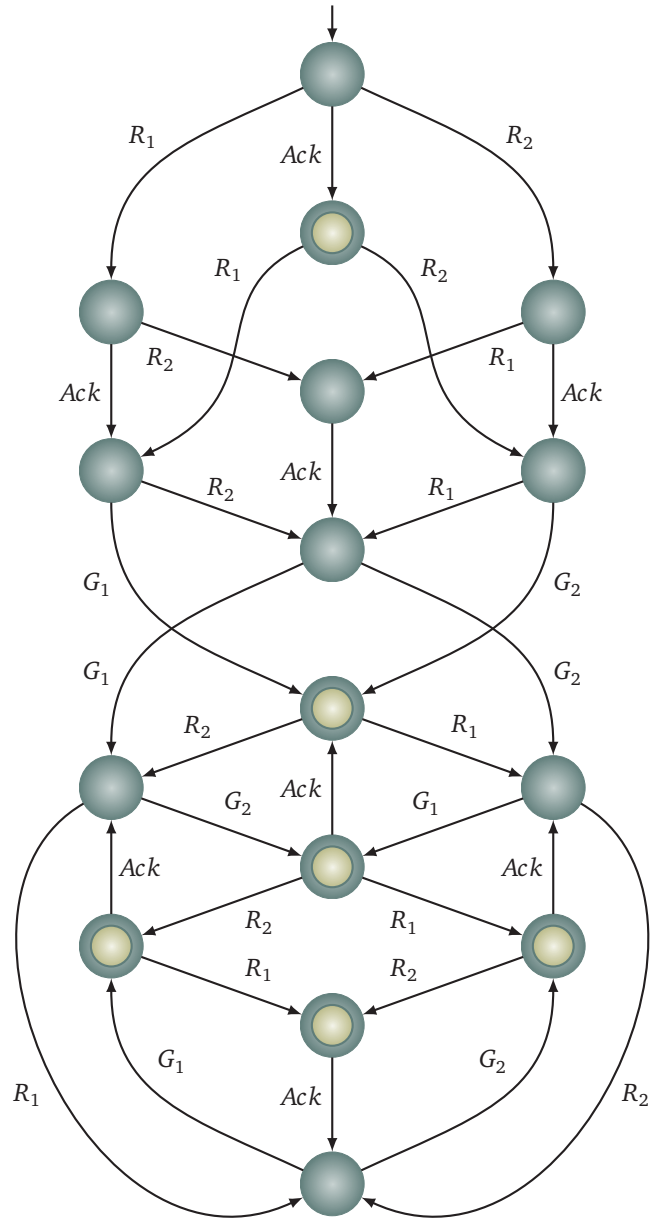


Figure 4.10: a DFA recognizing the traces that demonstrate the many ways a refinement of the sequencer can exceed the specification (cf. Equation 3.6, Equation 3.7, Figure 4.5 and Figure 4.9)

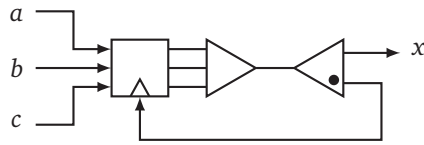


Figure 4.11: a majority gate with a bug in it (cf. Figure 2.10)

process combinator expression, but this effect can be understood as a consequence of the restrictive influence of the environment attached to it in S . Unlike the closed system S , the Petri net model of T alone contains open input transitions, which may fire therefore at any time, allowing the reachability graph to venture into spaces precluded by the environmental restrictions.

However, the real question motivating the present discussion is not whether T refines S , but how. In other words, what can T do that S can not? There is an unsupported assertion in Section 3.6.4 to the effect that premature acknowledgments are acceptable for T but not S . It is possible to substantiate to this claim in terms of trace recognizing automata. A recognizer for the intersection of Q_T , the quiescent trace set of T , with D_S , the divergent trace set of S , is shown in Figure 4.10. This DFA can be generated automatically by standard algorithms. To interpret this construction, we note that any trace in $Q_T \cap D_S$ is handled reliably by T but breaks S . For example, the trace $R_1 \text{ Ack } G_1$ leads to an accepting state of the DFA, being therefore appropriate for T but beyond the ability of S . The salient feature of any trace in this set is that an *Ack* input precedes the first grant.

4.5 Interim remarks

The informal introductory treatment of our subject draws to a close with this chapter, to be followed in Part II onwards with more math and less chat (or at least more math). The balance between ease of exposition and technical detail must shift toward the latter hereafter if interested readers are to acquire any genuine command of the material. Whatever its aesthetic appeal, this subject finds true vindication only as a tool in capable hands.

Nevertheless, the remaining chapters contained here should not be considered the last word. At this juncture, a rare confluence exists between a level of maturity in this subject and a near absence of significant competition for anyone with ambitions to capitalize on it either commercially or academically. For better or worse, this book can be expected to become obsolete in due course.

Hack toff

1. How can the reduced reachability graph in [Figure 4.4](#) be simplified further? What would an optimization algorithm have to do in general to cover this case? Would that be a good idea? (hint: [Section 6.5](#))
2. Consider these two processes.

$$X = \text{put } b$$

$$Y = \text{seq}(\text{put } b, \text{put } b)$$


Are they equivalent, and if not, which one refines the other? Demonstrate by manually constructing the Petri net models, (reduced) reachability graphs, transducers, and relational trace recognizers for both, with attention to any relevant conventions regarding degenerate cases. (hint: The answer is not a matter of opinion.)

3. The prime minister's old university chum fulfills a government DI circuit procurement contract with a block of wood, arguing that it meets every specification by virtue of its empty relational trace set (having read something like that in a book once [72]).
 - a) Is the statement technically correct that a process with an empty relational trace set meets every specification, or should it be qualified in some way?
 - b) What is the relational trace set of a block of wood and what class of specifications does it meet? (hint: Remember the alphabet.)
 - c) Ruled to be in breach of contract but no quitter he, our man makes amends by stapling one end of a labeled wire to the block for each input or output signal in the contract specification. Has his dialectical position improved?
4. The circuit designed by hand in [Figure 4.11](#) was proposed as an implementation of a 2-of-3 majority gate.
 - a) What is wrong with it?
 - b) How might the error be demonstrated in terms of the theory described in this chapter?
 - c) Suggest a methodology to detect errors of this nature as a matter of policy. Assume any transformation illustrated in [Figure 3.1](#) except technology mapping can be performed at no cost.
5. What would be a realistic example of a practical DI circuit that is clearly a compatible replacement for another but nevertheless does not meet the formal criteria for refinement? (hint: [item 4](#) and [item 5](#), page 488)

Part II

Formal Models

Everything is vague to a degree you do not realize till you have tried to make it precise.

Bertrand Russell

CHAPTER **5**

PETRI NET PLUMBING

A reasonable starting point to develop an understanding of DI circuits would be with a description of the primitive components from which all other DI circuits are to be built. Depending on the methodology, it is incumbent on the designer or the circuit synthesis tool developer to be familiar with them, because they form the basic abstraction layer above the underlying technology. Beyond certain minimal requirements, the range of possibilities in the choice of a set of primitives presents an engineering decision in itself. The approach favored in this book is to derive a standard assortment of useful modules from an extremely simple set of primitives.

This goal requires the entirety of Part II as a prerequisite due to the amount of technical detail involved, with our actual starting point being the development of a consistent style of specification for DI processes. The main purpose of this chapter is to establish a small core of process combinators to that end. This formalism not only enables a set of primitive components to be defined precisely in [Chapter 9](#), but enables high level behavioral circuit descriptions in a form suitable for automated synthesis as seen in Part IV.



As noted in Part I, Petri nets are well suited to specifying delay insensitive processes, so this chapter focuses on nothing but Petri nets. While drawing pictures of Petri nets is adequate for an introductory overview, taking the subject any further makes it unavoidable to have some way of writing them down and manipulating them formally. A bare minimum of mostly standard mathematical notation described in [Section 5.1](#) facilitates this task. [Section 5.2](#) takes a stab at a formal definition of Petri net-modeled DI processes. Subsequently a few algebraic operators proposed in [Section 5.3](#) allow certain very frequently used Petri net manipulations to be expressed without undue repetition or

verbosity. Finally, the rest of the chapter constructs seven process combinators explicitly in terms of Petri net models, which suffice for all process specifications of interest.

5.1 Mathematical conventions

Quite a few sets, functions, relations, tuples and related concepts show up in the sections to follow. These concepts are treated as fundamental, which is to say not derived from anything more basic. It would probably be a good idea for a reader completely unfamiliar with these terms to look them up somewhere (e.g., [163, 271]), but a quick primer covering a handful of relevant conventions should be enough for those having a nodding acquaintance with them.

5.1.1 Mapping

One way of operating on a set s is to transform every member $v \in s$ to a new entity $f(v)$ according to some function f . The result of this transformation is expressible as a set $(\mu f) s$ in terms of an operator μ defined as follows.

$$\mu = \lambda f. \lambda s. \bigcup_{v \in s} \{f v\} \quad (5.1)$$

We could think of μ as the operator that “maps” a function over any set. For example, a set $s = \{v_1, v_2, v_3\}$ could be transformed to $(\mu f) s = \{f v_1, f v_2, f v_3\}$. The result $(\mu f) s$ could have fewer members than s if f transforms more than one of them to the same result.

Technically [Equation 5.1](#) does not formally define μ as a function because it does not fully specify its domain, but an expression of the form μf does determine a function

$$(\mu f) : \mathcal{P}(d) \rightarrow \mathcal{P}(c)$$

for a known function $f : d \rightarrow c$ with domain d and codomain c , where $\mathcal{P}(d)$ denotes the set of all subsets of d , also known as its **power set**. Software practitioners might nevertheless tolerate μ as an example of a “polymorphic function”, suspending judgment on whether this usage can be justified. Like many to follow in this chapter, the lambda term in [Equation 5.1](#) may be regarded as a pseudo-code template for the computation of any member of a class of functions, forming the rigorous specification of a particular function only in combination with a concrete input type left to the reader’s discretion.

5.1.2 Domains and ranges

Any set s of pairs (a, b) can be regarded as a **relation**, making it meaningful to refer to its domain and range in terms of the μ operator defined by [Equation 5.1](#).

$$\mathcal{D}(s) = (\mu \lambda(a, b). a) s \quad (5.2)$$

$$\mathcal{R}(s) = (\mu \lambda(a, b). b) s \quad (5.3)$$

That is, the domain $\mathcal{D}(s)$ of a set s is the set of all left sides a of pairs (a, b) in the set, and the range $\mathcal{R}(s)$ is the set of all right sides b of pairs (a, b) in the set.

5.1.3 Cases

We often have to define functions by cases predicated on their arguments, but sometimes can do so more easily in terms of the Kronecker delta operator, which is defined conventionally as follows.

$$\delta_j^i = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \quad (5.4)$$

The i and j in this expression are normally taken to be natural numbers, but a useful “polymorphic” generalization is obtained by allowing any expressions for i and j whose comparison is meaningful.¹ For example, if s is a set, then the expression $\delta_{s \cup \{x\}}^s$ is unity whenever x is a member of s .

The Kronecker delta operator used in combination with an expression of the form $\langle a, b \rangle_k$ for $k \in \{0, 1\}$, makes it possible to define functions by cases. This expression is interpreted as

$$\langle a, b \rangle_k = \begin{cases} a & \text{if } k = 0 \\ b & \text{if } k = 1 \end{cases}$$



so that either term a or b is selected depending on whether or not i is equal to j in an expression of the following form.

$$(\lambda k. \langle a, b \rangle_k) \delta_j^i$$

The angle bracket notation introduced above pertains to a type of ordered sequence defined formally in [Chapter 7](#) and used more generally from [Chapter 8](#) onwards, but for purposes of this chapter and [Chapter 6](#) it can be regarded purely as a notational device.

5.1.4 Ordinals

The last notational device to be defined for now makes certain expressions shorter and far more easily readable than they would have to be without it. Let any **totally ordered** set s induce a **bijective** and **monotonic** function

$$s^\circ : s \mapsto \{n \in \mathbb{N} \mid n < |s|\}$$

denoted as shown. The expression $s^\circ(a)$ should be read as “the ordinal of a with respect to s ”, and is a number. With regard to terminology:

- A totally ordered set is one in which every member either precedes or follows every other member according to some known ordering.
- A bijective function is another term for a one-to-one function, meaning that every argument in its domain is mapped to a unique result in its codomain, and every result in its codomain has some argument in the domain that maps to it.
- A monotonic function always maps a lesser argument to a lesser result than that of a greater argument according to whatever ordering is relevant to the context.
- The notation \mapsto above is used instead of \rightarrow to emphasize that a function is either bijective or **injective**. An injective function also maps every argument to a unique result, but some members of its codomain might not have any argument in the domain mapped to them.

¹May Leopold Kronecker forgive us.

It is possible to give a precise definition of the ordinal function as

$$s^\circ = \lambda x. \log_2 |\{p \in \mathcal{P}(s) \mid x = \max(p)\}| \quad (5.5)$$

if the maximum of any set p with respect to the ordering is given by $\max(p)$, because luckily the cardinality of $\{p \in \mathcal{P}(s) \mid x = \max(p)\}$ always lands on a power of two. Explicitly specifying the order is mostly unnecessary in this chapter, although there is more to say about it in [Chapter 6](#). For subsets s of \mathbb{N} , the usual ordering on natural numbers applies. For example, a set $s = \{3, 7, 9\}$ satisfies $s^\circ(3) = 0$, $s^\circ(7) = 1$, and $s^\circ(9) = 2$. Furthermore, because s° is bijective, it is meaningful to express its inverse as

$$s^{\circ^{-1}} : \{n \in \mathbb{N} \mid n < |s|\} \rightsquigarrow s.$$

5.2 From Petri nets to processes

To upgrade our understanding of Petri nets from the pictorial to something more substantial, we can start by ignoring superfluous information about the way they are depicted and focusing on what matters. What places and transitions are in the Petri net, how are they all connected, and what places are initially marked? Any reasonable model of a Petri net could be expected to determine at least a set P of places and a set T of transitions. Then it should not be too far fetched to envision a set $M \subseteq P$ containing the places that are initially marked, unless already this step is overly presumptuous. Could a place ever hold more than one token, for example if the Petri net is unsafe? In that case the marking might need to be modeled by a function $M : P \rightarrow \mathbb{N}$ such that $M(p)$ indicates the number of tokens in a place p . Could there be more than one type of token, for example in a colored Petri net? Maybe \mathbb{N} is not big enough. The possibilities are endless.

As for the connections from places to transitions and *vice versa*, a relation $A \subseteq (P \times T) \cup (T \times P)$ containing a pair (a, b) for each arc from a vertex a to a vertex b would seem sufficient, unless this too is an oversimplification. Maybe some arcs are more important than others so some sort of a weighting function should be involved. Maybe an adjacency matrix would be more convenient than any weighting function or relation in this form so that we can apply matrix algebra to it (assuming we know which place or transition corresponds to each row or column).

The point of these digressions is that there is no right way to build a concrete model of a Petri net, only the way that best suits the intended application. This activity is more like provisioning a tool box than enacting legislation, so it should come as no surprise if the proposals in this section are similar but not identical to those of other sources. Fortunately the Petri nets needed for our purposes are toward the simple-minded end of the spectrum.

5.2.1 A concrete model

To get down to business, a set P of places, a set T of transitions, a set $M \subseteq P$ of initially marked places, and a set $A \subseteq (P \times T) \cup (T \times P)$ of arcs with the interpretation above will do nicely, and there is no compelling reason for them not to feature explicitly as such in the concrete model. It may be recalled from [Chapter 3](#) that a set of finally marked places also has a role to play. Although no final marking is explicitly represented in most conventional Petri net models,²



²but see Petri nets with Boundaries [235] or Petri Box Calculus and related work [25, 26] for something similar

there is no impediment to including one as a set $F \subseteq P$. Without getting too creative, we can model a Petri net as a tuple

$$(P, T, A, M, F) \in \mathbb{P}$$

where \mathbb{P} is the universe of tuples of this form, but for this statement to make sense, something should also be said about the universes from which P and T are drawn. We return to this point shortly.

It may also be recalled from [Chapter 3](#) that some transitions in the Petri net are to represent externally observable signals, while others are hidden or anonymous, but the model hitherto does nothing to express this distinction. A simple remedy is to regard observable transitions as members of a universe \mathbb{T} , and the rest as members of a universe \mathbb{V} disjoint from \mathbb{T} . Then for a Petri net with transitions T , we write $T \cap \mathbb{T}$ for the observable transitions, $T \cap \mathbb{V}$ for the unobservable transitions, and just T in contexts where the distinction is immaterial. This convention implies in general that the transitions T in any member of \mathbb{P} belong to the universe $\mathbb{T} \cup \mathbb{V}$.

No such distinction is necessary for places in a Petri net, which are always unobservable. If we require $P \subseteq \mathbb{V}$ to hold for the set of places P in any member of \mathbb{P} , then the issue mentioned above is resolved by defining \mathbb{P} as the following product.

$$\mathbb{P} = \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times (\mathbb{T} \cup \mathbb{V})) \times \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \quad (5.6)$$

However, not every member of \mathbb{P} by this definition makes sense as a Petri net. To be extra precise, a tuple $(P, T, A, M, F) \in \mathbb{P}$ is called **well formed** if it satisfies these conditions

- $P \cap T = \emptyset$
- $M \cup F \subseteq P$
- $A \subseteq (P \times T) \cup (T \times P)$
- $T \cap \mathbb{V} \subseteq \mathcal{D}(A) \cap \mathcal{R}(A)$
- $\forall t \in T \cap \mathbb{T}. \mathcal{R}(A \cap (\{t\} \times \mathbb{V})) \ominus \mathcal{D}(A \cap (\mathbb{V} \times \{t\})) \neq \emptyset$

where $x \ominus y$ denotes the symmetric set difference operation $(x - y) \cup (y - x)$ or equivalently $(x \cup y) - (x \cap y)$. The first three conditions mean every vertex is either a place or a transition, only places can be marked, and arcs connect only places to transitions or *vice versa*. The fourth means that unobservable transitions must be both the origin and the terminus of at least one arc (*i.e.*, not “open”), and the last implies that observable transitions must be at least one or the other. The last two conditions are not quite as essential but still a sign that something is amiss if they do not hold. In particular, without them an observable transition could fire uncontrollably with no safety violation. We allow \mathbb{P} to contain non-well formed Petri nets because they are convenient in computations.

5.2.2 Presets and postsets

We may note in passing that another way of expressing the last condition above uses the more conventional **preset** and **postset** notations $\bullet t$ and $t \bullet$.

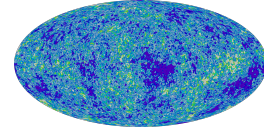
$$\forall t \in T \cap \mathbb{T}. t \bullet \ominus \bullet t \neq \emptyset$$

For an individual vertex v , whether a place or transition, the postset $v\bullet \subset \mathbb{T} \cup \mathbb{V}$ is defined as the set of vertices at the termini of its outgoing arcs, and the preset $\bullet v$ is defined as the set of vertices whose postsets contain v . If v is a set of vertices, then $v\bullet$ denotes the union of all sets $t\bullet$ for members $t \in v$, and $\bullet v$ is analogous.

This notation has the advantage of being widely used in the Petri net literature and being independent of the concrete representation, but the disadvantage of being ambiguous if a vertex v belongs to more than one Petri net in the same context, or even if v belongs to just one Petri net but the usage is something like $\bullet v\bullet$. The notation is therefore confined mostly to informal discussions in this book as an aid to intuition except in [Chapter 9](#) where it proves irresistibly accommodating.

5.2.3 Hacking the universe

As noted above, this theory already requires the set of unobservable vertices \mathbb{V} to be disjoint from the set of observable vertices \mathbb{T} , but a few further assumptions about them make life easier with no downside. Taking \mathbb{V} to be countably infinite and totally ordered justifies the use of functions \mathbb{V}° and $\mathbb{V}^{\circ-1}$ for tapping an unlimited supply of distinct vertices. For example, if $v = P \cup (T \cap \mathbb{V})$ is the set of unobservable vertices in a Petri net (P, T, A, M, F) , then a new vertex that differs from all of them is given by $\mathbb{V}^{\circ-1}(1 + \mathbb{V}^\circ \max v)$.



At a stretch, we might also deem \mathbb{T} to be countably infinite, but the imposition of a universal total ordering on it would be artificial and awkward in practice because \mathbb{T} is envisioned to contain tangible objects (*i.e.*, the terminals interfacing a process with its environment). A pair of weaker assumptions is sometimes invoked as a workaround. An injective function $\eta : S \rightarrow \mathbb{N}$ mapping terminals to numbers can be fixed temporarily for any finite subset $S \subset \mathbb{T}$, and another arbitrary unspecified function $\gamma : \mathcal{P}(\mathbb{T}) \rightarrow \mathbb{T}$, not necessarily injective, takes any finite subset S of \mathbb{T} to a non-member thereof, $\gamma S \in \mathbb{T} - S$. Formally γS is a terminal, but it should be regarded intuitively as a placeholder for a terminal used only in calculations.

5.2.4 Open Petri nets

The Petri net model considered up to this point makes a bit of progress toward specifying a DI process by capturing its observable behavior precisely. It also provides a hook for stringing multiple DI processes together in sequence by explicitly indicating their termination conditions via the final marking. However, the job is not yet done. Any DI process also has an input alphabet $I \subset \mathbb{T}$ and an output alphabet $O \subset \mathbb{T}$ associated with it, and it would be better to have them written down somewhere than to try to remember them.

One could argue that the alphabets can always be inferred from a Petri net model by inspecting its observable transitions $T \cap \mathbb{T}$. Doing so might yield the union $I \cup O$ of the alphabets but not identify either of them unambiguously. Based on numerous illustrations in [Chapter 3](#), it might be countered that the ambiguity is easily resolved insofar as any input transition $i \in I$ has an empty preset $\bullet i = \emptyset$, and any output transition $o \in O$ has an empty postset $o\bullet = \emptyset$. It is not a formal requirement that every observable transition in a Petri net must have at least one or the other set empty, but we could make it one by defining

$$\hat{\mathbb{P}} = \{(P, T, A, M, F) \in \mathbb{P} \mid \mathcal{D}(A) \cap \mathcal{R}(A) \cap \mathbb{T} = \emptyset\} \quad (5.7)$$

as the universe of *open Petri nets*. However, a blanket restriction to open Petri nets would be onerous for both theoretical and practical reasons (*e.g.*, difficulties defining the **env** combinator and



Figure 5.1: A permanently disabled/unsafe input transition a is expressible as such in an open Petri net by an unmarked preset place disable it (left) or by a marked postset place to make it unsafe (right), with either alternative being semantically equivalent to the other.

more costly enumeration of reachability graphs), and the rule proposed above for distinguishing inputs from outputs admits exceptions even then (e.g., Figure 5.1).

5.2.5 Process models

The takeaway from this discussion is that a record of the input and output alphabets of a DI process must be kept independently of the Petri net model. To ensure that process alphabets are recorded along with other aspects of their specifications, let the set \mathbb{D} of **Petri net-modeled DI processes** be that of all tuples $(I, O, N) \in \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T}) \times \mathbb{P}$ meeting these conditions.

- $I \cap O = \emptyset$
- N is well formed.
- $T \cap \mathbb{T} \subseteq I \cup O$ where $(P, T, A, M, F) = N$ is the Petri net model in the tuple (I, O, N) .

A subset $\tilde{\mathbb{D}} \subset \mathbb{D}$ restricted to open Petri net models nevertheless has its uses, so it is defined as

$$\tilde{\mathbb{D}} = \{(I, O, N) \in \mathbb{D} \mid N \in \tilde{\mathbb{P}}\} \quad (5.8)$$

for a universe $\tilde{\mathbb{P}} \subset \hat{\mathbb{P}} \subset \mathbb{P}$ of Petri nets that are not only open, but meet the additional condition that all presets and postsets of their observable transitions are mutually disjoint.

$$\tilde{\mathbb{P}} = \left\{ (P, T, A, M, F) \in \hat{\mathbb{P}} \mid \left(\lambda p. \delta_{\bigcup_{p_l} |s_l|} \right) \bigcup_{t \in \mathbb{T}} \{ \mathcal{D}(A \cap (\mathbb{V} \times \{t\})), \mathcal{R}(A \cap (\{t\} \times \mathbb{V})) \} = 1 \right\} \quad (5.9)$$

This condition turns out to be a matter of technical convenience for defining process combinators and incurs no loss of generality because any Petri net has a behavioral equivalent of this form.

The main benefit of requiring disjoint alphabets and well formed Petri nets by definition of the universe \mathbb{D} (and by implication $\tilde{\mathbb{D}}$) is to avoid repeatedly having to say what happens when a function of \mathbb{D} is applied to a tuple (I, O, N) where the alphabets intersect or N is not well formed, because it is implicitly undefined. In either definition, we allow $T \cap \mathbb{T} \subseteq I \cup O$ instead of demanding $T \cap \mathbb{T} = I \cup O$ in case the Petri net N can be consequently simplified, because any possible way of simplifying the Petri net without altering its semantics is a net win.

This last relaxation necessarily allows some members of the alphabet not to appear as transitions in the Petri net, and raises the question of how these missing signals should be interpreted. The short answer is that any input signal absent from the Petri net is always unsafe, and any output signal absent from the Petri net is never emitted. A longer answer follows from the semantic interactions of the input and output completion operators to be defined presently in Section 5.3, and their effects on the trace semantics following from the constructions of the reachability graph in Chapter 6 and the transducer model in Chapter 7.

5.3 Editing operations

Having a few intuitively clear but rigorously defined operations on Petri nets at our disposal makes the job of building interesting and useful Petri net-modeled DI processes run smoothly. Some operations relevant to visibility and scope are developed in [Section 5.3.1](#), operations similar to the usual binary operations on sets are developed in [Section 5.3.2](#) and [Section 5.3.3](#), and a few more designed to make alphabets easier to handle are developed in [Section 5.3.4](#).

5.3.1 Rewriting

A unary operator and three binary operators introduced in this section are useful for Petri net transformations. A general way of consistently rewriting the vertices in a Petri net enables coalescence of anonymous vertices and anonymization of observable vertices. The last operator helps to prevent anonymous vertices from clashing when two Petri nets are combined.



Mapping

Just as the μ operator defined by [Equation 5.1](#) maps a function over a set, a concept of mapping a function over a Petri net is also useful. If a function $f : \mathbb{T} \cup \mathbb{V} \rightarrow \mathbb{T} \cup \mathbb{V}$ transforms a vertex v to a vertex $f v$, then we can envision $f^\bullet : \mathbb{P} \rightarrow \mathbb{P}$ as a function that transforms all occurrences of v throughout a Petri net $X \in \mathbb{P}$ consistently to $f v$ in the Petri net $f^\bullet X$. A definition scoring no points for subtlety is as follows.

$$f^\bullet(P, T, A, M, F) = ((\mu f) P, (\mu f) T, (\mu (\lambda(a, b). (f a, f b))) A, (\mu f) M, (\mu f) F) \quad (5.10)$$

Note that f need not be injective, so it could happen that distinct vertices v and w are mapped to the same vertex $f v = f w$. In this case, the application of f^\bullet to a Petri net X coalesces them in $f^\bullet X$, which can be useful when done deliberately. Although it is not needed until [Chapter 8](#), a related notation

$$f^\diamond(I, O, N) = ((\mu f) I, (\mu f) O, f^\bullet N) \quad (5.11)$$

is useful for expressing a change of symbols mapped over a whole process $X = (I, O, N) \in \mathbb{D}$.

Coalescence

An easy way of expressing the idea of coalescence in general is by X/S , read “ X coalesced by S ”. If $X \in \mathbb{P}$ is a Petri net and $S \in \mathcal{P}(\mathcal{P}(\mathbb{V}))$ is a set of mutually disjoint sets of vertices, then X/S is a Petri net derived from X by rewriting every vertex $v \in p$ in every set $p \in S$ to the same member of p . Non-members of $\bigcup S$ are the same in X/S as in X .

It is usually adequate to rewrite a vertex $v \in p$ to the minimum member of p , where p is a member of S , but an explicit provision is appropriate for the case when some but not all members of p are initially marked (*i.e.*, members of M with respect to the Petri net $X = (P, T, A, M, F)$ in the expression X/S). The issue is whether the members of p should be coalesced into a marked place in X/S or an unmarked place.

There is a good reason to prefer one convention over the other. One of the uses for this operation is in parallel composition. As shown in [Figure 3.11](#), when the same input transition appears in both

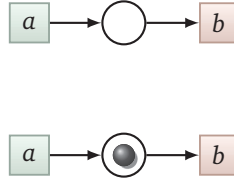


Figure 5.2: The coalescence of a marked place with an unmarked place should be a marked place so that the parallel composition of these two processes is equivalent to the lower one.

operands to be composed in parallel, the places adjacent to it are coalesced, with the intuition being that an input signal can propagate either way. With regard to the parallel composition of the two processes depicted in Figure 5.2, an input signal initially could propagate safely to the top one or unsafely to the bottom one, so it is initially unsafe. With no initial input signal, an output signal eventually appears, after which the initially marked place is unmarked and an input is safe. This behavior is indistinguishable from that of the lower process by itself. The coalescence operator would naturally do the right thing if combining an unmarked place with a marked place were to result in a marked place.

To describe this operation more formally, we have the set $p \in S$ containing a vertex v given by

$$p = \bigcup \{l \in s \mid v \in l\}$$

assuming there is one and the members of S are pairwise disjoint. Having found this set, we can then ask whether it intersects M , the set of initially marked places, and express its restriction to M in that case as

$$(\lambda j. \langle p \cap M, p \rangle_j) \delta_{\emptyset}^{p \cap M}$$

to describe the desired image of the vertex v under the transformation as either the minimum member of p or the minimum marked member of p as appropriate

$$\min (\lambda j. \langle p \cap M, p \rangle_j) \delta_{\emptyset}^{p \cap M}$$

whenever v is a member of p , but as v itself otherwise.

$$(\lambda i. \langle v, \min (\lambda j. \langle p \cap M, p \rangle_j) \delta_{\emptyset}^{p \cap M} \rangle_i) \delta_{\{v\}}^{\{v\} \cap p}$$

Summarizing this result as $N_0(X, S) v$ in terms of a function $N_0 : \mathbb{P} \times \mathcal{D}(\mathcal{D}(\mathbb{V})) \rightarrow (\mathbb{V} \rightarrow \mathbb{V})$ given by

$$N_0 = \lambda((P, T, A, M, F), S). \lambda v. (\lambda p. (\lambda i. \langle v, \min (\lambda j. \langle p \cap M, p \rangle_j) \delta_{\emptyset}^{p \cap M} \rangle_i) \delta_{\{v\}}^{\{v\} \cap p}) \bigcup \{l \in s \mid v \in l\}$$

we have the following definition for the coalescence operator based on Equation 5.10.

$$X/S = N_0(X, S) \star X \tag{5.12}$$

Anonymization

Another useful operation can also be motivated by the example of parallel composition shown in Figure 3.11. When an observable transition is an input to one and an output from the other operand, it should be made anonymous in the result. A Petri net $X \in \mathbb{P}$ resulting from a parallel composition and a set of observable transitions $S \in \mathcal{P}(\mathbb{T})$ common to the operands as inputs to one and outputs from the other should be transformed to $X \setminus S$, read “ X anonymized by S ”, by transforming each member of S to an anonymous transition wherever it occurs in X .

The only technical difficulty is ensuring that the anonymous vertices created for this occasion do not clash with any already present in the Petri net X . That is, they should all be non-members of $N_1 X$ for a function $N_1 : \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V})$ defined by

$$N_1 = \lambda(P, T, A, M, F). (\lambda s. \langle s, \{\min \mathbb{V}\} \rangle_{\delta_s^\emptyset}) ((P \cup T) \cap \mathbb{V}) \quad (5.13)$$

taking a Petri net $X \in \mathbb{P}$ to the minimum non-empty set containing its anonymous vertices. For this purpose we need to invoke the assumption of a function $\eta : S \rightarrow \mathbb{N}$ taking any member t of S to a unique natural number $\eta t \in \mathbb{N}$ as discussed in Section 5.2.3. Then we can associate an anonymous transition

$$\mathbb{V}^{\circ-1}(1 + (\eta t) + \mathbb{V}^\circ \max N_1 X)$$

with each $t \in S$, which is certain not to clash with any in X because its ordinal exceeds theirs. In general for an arbitrary vertex t that may or may not be a member of S , the desired image under the transformation is

$$(\lambda i. \langle \mathbb{V}^{\circ-1}(1 + (\eta t) + \mathbb{V}^\circ \max N_1 X, t \rangle_i) \delta_s^{S-\{t\}}$$

which is to say that non-members of S are invariant. Taking this expression to induce a function of t and mapping it over X according to Equation 5.10 yields the full specification.

$$X \setminus S = (\lambda t. (\lambda i. \langle \mathbb{V}^{\circ-1}(1 + (\eta t) + \mathbb{V}^\circ \max N_1 X, t \rangle_i) \delta_s^{S-\{t\}})^\bullet X \quad (5.14)$$

Separation

The last operator concerned primarily with vertex rewriting is perhaps more difficult to visualize but at least as useful as those above. When Petri nets $X, Y \in \mathbb{P}$ are to be combined by parallel composition or some other operation, the result normally includes vertices from both. Whereas the observable vertices they have in common may be fused or anonymized, nothing meaningful is implied by an anonymous vertex $v \in \mathbb{V}$ appearing in both X and Y . To avoid mistakenly fusing the two copies of it, we could apply a function f^\bullet to one of X or Y that moves the anonymous vertices they have in common out of the way in preparation to combine them without clashes. That is, if v is an anonymous vertex in X and also in Y , then its image $f v \in \mathbb{V}$ should be one that belongs to neither X nor Y . A function f suitable for this purpose would have to be tailored specifically to X and Y . Let such a function be known as the **separation function** of X and Y and denoted $X \wr Y$.

To define a separation function formally, let $x = N_1 X$ denote the set of anonymous vertices in the Petri net $X \in \mathbb{P}$, and similarly let $y = N_1 Y$ denote those of Y by Equation 5.13. Then to avoid any clashes, a vertex $v \in x \cap y$ common to both could be mapped to a new vertex

$$\mathbb{V}^{\circ-1}(1 + ((x \cap y)^\circ v) + \mathbb{V}^\circ \max (x \cup y))$$



whose ordinal exceeds that of any in $x \cup y$ and differs from those of other members of $x \cap y$. Expressing this result as $N_2(X, Y) \nu$ in terms of a function $N_2 : \mathbb{P} \times \mathbb{P} \rightarrow (\mathbb{V} \rightarrow \mathbb{V})$ defined by

$$N_2 = \lambda(X, Y). (\lambda(x, y). \lambda v. \mathbb{V}^{\circ-1}(1 + ((x \cap y)^\circ \nu) + \mathbb{V}^\circ \max(x \cup y))) (N_1 X, N_1 Y)$$

we have most of the separation function already. To maintain invariance of vertices ν outside of the intersection $s = x \cap y$ under the transformation, we need only generalize this result to

$$(\lambda i. \langle N_2(X, Y) \nu, \nu \rangle_i) \delta_s^{s-\{v\}}$$

in the definition overall.

$$X \wr Y = (\lambda s. \lambda v. (\lambda i. \langle N_2(X, Y) \nu, \nu \rangle_i) \delta_s^{s-\{v\}}) ((N_1 X) \cap (N_1 Y)) \quad (5.15)$$

5.3.2 Sums

In this section we investigate what a binary operation like union might mean for Petri nets, starting from a naive concept and progressing to a more sophisticated one.

Separated sums

The simplest analog to the union of a pair of sets for a pair of Petri nets $X, Y \in \mathbb{P}$ would be their componentwise union $N_3(X, Y)$ given by a function $N_3 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ defined as

$$N_3 = \lambda((P, T, A, M, F), (P', T', A', M', F')). (P \cup P', T \cup T', A \cup A', M \cup M', F \cup F')$$

By itself this operation is not useful in practice because it could yield differing results for semantically equivalent operands depending on whether their anonymous vertices clash. A minimally useful form of Petri net union would invoke the separation function defined by [Equation 5.15](#).

$$X \uplus Y = N_3(X, (X \wr Y) \blacklozenge Y) \quad (5.16)$$

This operation, denoted as shown and called a **separated sum** hereafter, fuses the visible vertices but segregates the anonymous ones according to their provenance.

Coalesced sums

A more sophisticated form of Petri net union than the separated sum could support the desired semantics of parallel composition and other process combinators better. As noted in [Section 3.6](#), the most useful thing to happen by default when processes are composed is for their common outputs to be merged and their common inputs arbitrated. This behavior finds a natural expression in terms of a Petri net model when places adjacent to the terminals common to both operands are coalesced, as shown in [Figure 3.11](#). The less desirable alternative of keeping them separate results in a Petri net that behaves as if common inputs are broadcast to both operands and common outputs are barrier synchronization points.

A straightforward remedy is to coalesce the preset $\bullet t$ and the postset $t \bullet$ of every observable transition $t \in T \cap \mathbb{T}$ in the Petri net $Z = (P, T, A, M, F)$ resulting from a separated sum $X \uplus Y$. The set

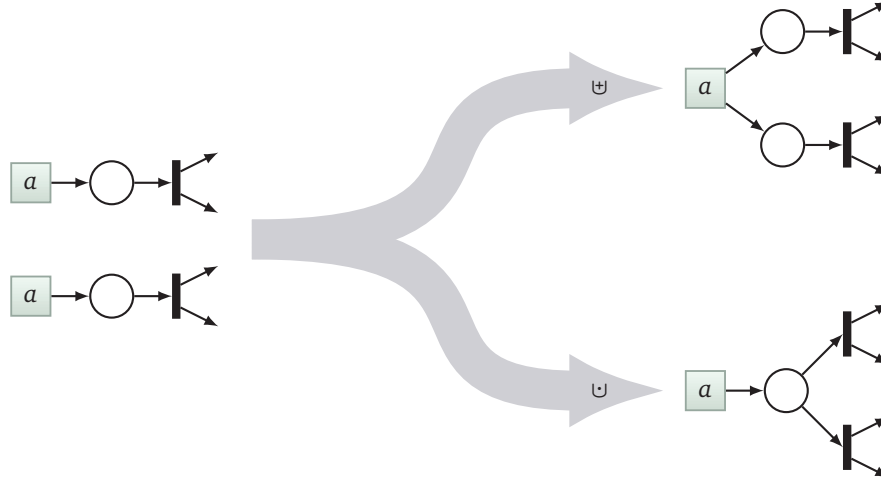


Figure 5.3: A separated sum of two Petri nets with an input a in common fuses only the input (above), while a coalesced sum also fuses its adjacent places (below).

of all relevant presets and postsets is easily constructed in terms of a function $N_4 : \tilde{\mathbb{P}} \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{V}))$ defined by

$$N_4 = \lambda(P, T, A, M, F). \bigcup_{t \in T \cap \mathbb{T}} \{ \mathcal{D}(A \cap (\mathbb{V} \times \{t\})), \mathcal{R}(A \cap (\{t\} \times \mathbb{V})) \}$$

so that a Petri net behaving more in keeping with expectations can be expressed

$$W = (\lambda Z. Z / (N_4 Z)) (X \uplus Y)$$

using the coalescence operator defined by Equation 5.12. Note that this result is undefined unless Z is a member of $\tilde{\mathbb{P}}$, because only then are the members of $N_4 Z$ mutually disjoint as required by the coalescence operator.

This coalescence operation is invariably followed by another when Petri nets are combined, which is to anonymize all of the hitherto observable transitions that have become hidden from the environment due to being connected on both sides. These transitions are easily identified as members of $\mathcal{D}(A) \cap \mathcal{R}(A) \cap \mathbb{T}$, where A is the set of arcs in the result $W = (P, T, A, M, F)$. An enhanced Petri net union operator bundling both the coalescence and anonymization steps therefore can be defined by

$$X \uplus Y = (\lambda W. (\lambda(P, T, A, M, F). W \setminus (\mathcal{D}(A) \cap \mathcal{R}(A) \cap \mathbb{T})) W) (\lambda Z. Z / (N_4 Z)) (X \uplus Y) \quad (5.17)$$

in terms of Equation 5.14 and denoted as shown. This operation is called a **coalesced sum** hereafter. Figure 5.3 illustrates the difference between separated and coalesced sums at least as far as coalescence is concerned.

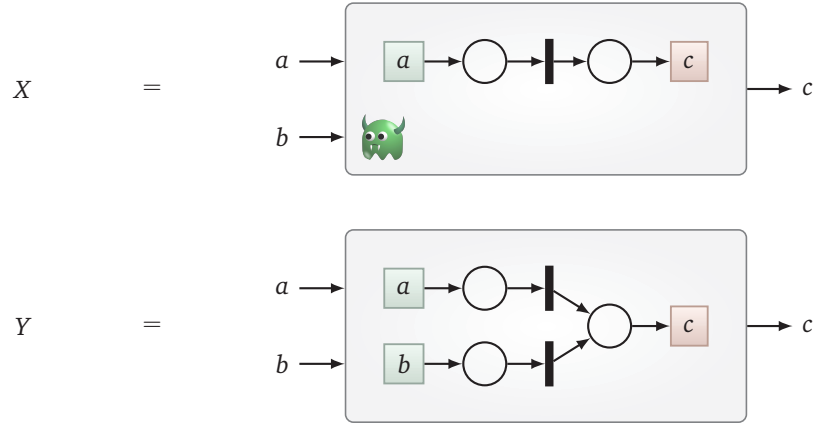


Figure 5.4: Two processes $X = (I, O, N)$ and $Y = (I, O, N')$ have the same alphabets and different Petri net models. X diverges on an input of b because b is not a transition in its Petri net model N .

5.3.3 Differences

An analog to the set difference operation is mostly straightforward to extend to Petri nets $X, Y \in \mathbb{P}$ as $N_5(X, Y)$ in terms of a function $N_5 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ specifying their componentwise difference.

$$N_5 = \lambda((P, T, A, M, F), (P', T', A', M', F')). (P - P', T - T', A - A', M - M', F - F')$$

To make this operation a bit more convenient, we can make it more conducive to a well formed result given a well formed left operand by restricting the arcs and markings to refer only to surviving vertices.

$$X \div Y = (\lambda(P, T, A, M, F). (P, T, A \cap ((P \times T) \cup (T \times P)), M \cap P, F \cap P) N_5(X, Y) \quad (5.18)$$

This operation does not need a special name other than perhaps the Petri net difference, but is denoted hereafter as shown.

5.3.4 Completion

As noted in [Section 5.2.5](#), the alphabets I and O are specified separately from the Petri net model N in a triple $(I, O, N) \in \mathbb{D}$ to decouple the interface from the behavioral specification, so the situation could arise where a member of the input alphabet I is not a transition in N . An intuitive case could be made that such an input would be silently ignored because it is not connected to anything inside the box, but a simpler semantics follows if we allow a more radical interpretation. When a process specification provides no information about the effect of an input, the most reasonable assumption is not that the input is silently ignored, but that there is nothing further to assume, or in other words that the process is rendered wholly unpredictable by the input.

This interpretation fits with the idea that the missing input, viewed as a Petri net transition, is never enabled. The only other way for an input transition to be disabled is for it to have a preset containing an unmarked place as shown at the left of [Figure 5.1](#). This configuration could appear

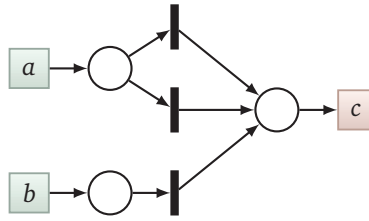


Figure 5.5: the coalesced sum $N \cup N'$ of the Petri nets from Figure 5.4

in a closed Petri net specified by the `env` combinator to indicate that this input is not needed, so the specification imposes no requirements about its effects. (See Section 3.6.4 for motivation.) Having the same interpretation for disabled input transitions in open and closed Petri nets causes less trouble all around.

We may often loosely refer to such an input as “prohibited” for lack of a better term, but it should be intuitively clear that no physically realistic process or its designer can ever literally prohibit the environment from transmitting an input signal to it.³ A more appropriate adjective, if it existed, would describe these inputs not exactly as prohibited but as inputs with unknown and presumptively displeasing effects.

The reason for dwelling on this point is to cope effectively with processes having prohibited inputs, for which it is not always valid to obtain the Petri net model of their parallel composition simply as the coalesced sum of their respective Petri net models. The rest of this section suggests a way of encapsulating this knowledge in terms of three so called completion operators on Petri nets and alphabets.

Input completion

The example in Figure 5.4 motivates the idea of input completion. There are two processes $X, Y \in \tilde{\mathbb{D}}$ both with an input alphabet $\{a, b\}$ and an output alphabet $\{c\}$, and for reasons explained above, X diverges given an input of b whereas Y acknowledges either input. In a parallel composition of these two processes, an input signal of b could propagate to either of them. If it goes to X , then the system diverges, so it is appropriate to regard b as unsafe for the combined system even though it would be safe for Y alone.

One might normally expect the Petri net model of a parallel composition to be something like the coalesced sum of the operands’ Petri net models, but in this example that would be as shown in Figure 5.5, which is behaviorally equivalent to the Petri net model N' of Y . This result misrepresents the process semantics because it gives no indication that b is unsafe.

This problem could be avoided by modifying the Petri net model N of X to reflect the complete input alphabet $\{a, b\}$ in its transitions as shown in Figure 5.6. Then by Equation 5.12 and Equation 5.17, the postset place for b in the coalesced sum would be marked, indicating that an input of b is unsafe. This remedy is always effective when N is a member of $\tilde{\mathbb{P}}$ so that the postsets of its observable transitions are mutually disjoint. It might not always be necessary, but let us hold that thought.

³Processes subject to this limitation, as opposed to those that magically are not, used to be called *receptive* [72, 128].

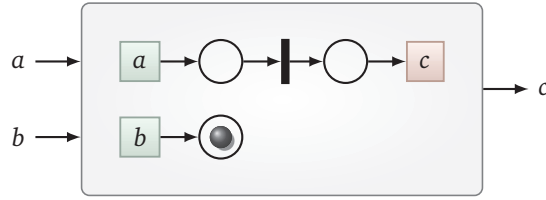


Figure 5.6: Input completion of N with $\{b\}$ makes the prohibition of b explicit.

To proceed along these lines, let the expression $N \triangleleft S$ denote a Petri net $N = (P, T, A, M, F) \in \tilde{\mathbb{P}}$ with all input transitions $t \in S$ added to it that are missing from T , and a separate marked postset place for each of them. Each arc in a set

$$a = (\mu \lambda t. (t, \mathbb{V}^{\circ-1} \eta t)) (S - T)$$

has a member $t \in S - T$ at its origin and the place $\mathbb{V}^{\circ-1} \eta t$ at its terminus, the latter being arbitrarily selected to have the ordinal ηt by an injective function $\eta : S \rightarrow \mathbb{N}$ assumed to be fixed for this purpose. (See Section 5.2.3.) A Petri net assembling all of the necessary places, transitions, arcs, and markings to be added to N as a tuple is then

$$(\mathcal{R}(a), \mathcal{D}(a), a, \mathcal{R}(a), \emptyset)$$

so a definition for input completion can be summarized as follows.

$$N \triangleleft S = (\lambda(P, T, A, M, F). N \uplus (\lambda a. (\mathcal{R}(a), \mathcal{D}(a), a, \mathcal{R}(a), \emptyset)) (\mu \lambda t. (t, \mathbb{V}^{\circ-1} \eta t)) (S - T)) N \quad (5.19)$$

Output completion

Notably there is no need for a corresponding operation in the way of output completion as far as parallel composition is concerned. An output transition missing from a Petri net means the process never emits that output. The parallel composition of two processes can output whatever either of them can, and this behavior is exhibited automatically by the coalesced sum of their respective Petri net models.

However, output completion may be appropriate for a process representing the environment of another process in the context of the **env** combinator. Every input transition on the latter process needs a matching output transition from the environment, even if only to disable it. Otherwise, the input is left open and always enabled.

The output completion $N \triangleright S$ for a Petri net $N = (P, T, A, M, F) \in \tilde{\mathbb{P}}$ and a set $S \subset \mathbb{T}$ represents a Petri net similar to N but with permanently disabled open output transitions $S - T$ added to it. This effect is easily achievable by the transitions in $S - T$ each having an unmarked place in their preset. The definition is similar to Equation 5.19 but with the arcs oppositely directed.

$$N \triangleright S = (\lambda(P, T, A, M, F). N \uplus (\lambda a. (\mathcal{D}(a), \mathcal{R}(a), a, \emptyset, \emptyset)) (\mu \lambda t. (\mathbb{V}^{\circ-1}(\eta t), t)) (S - T)) N$$

Mutual input completion

Any process $(I, O, N) \in \tilde{\mathbb{D}}$ always has an equivalent $(I, O, N \triangleleft I)$ wherein every member of the input alphabet is explicitly represented in the Petri net model, so converting it to this form before composing it with anything would mean never having to worry about prohibited inputs. However, a more parsimonious alternative may be possible depending on the process (I', O', N') with which it is to be composed. If an input $t \in I$ is prohibited according to N (i.e., absent from its set of transitions) but also prohibited according to N' , then there is no need for the input completion operand to include t because it would still be prohibited in the coalesced sum $N \cup N'$. In other words, rather than $N \triangleleft I$, only the simpler Petri net $N \triangleleft I \cap \nu N'$ is needed, where $\nu N'$ is the set of transitions T associated with N' by a function

$$\nu = \lambda(P, T, A, M, F). T.$$

Similarly, the Petri net N' of the other process requires only an input completion by $I' \cap \nu N$.

To take advantage of these observations, we need to know something about both Petri nets $N, N' \in \tilde{\mathbb{P}}$ and their respective input alphabets $I, I' \in \mathcal{P}(\mathbb{T})$, so a concept of **mutual input completion** as a binary operation like $(N, I) \bowtie (N', I')$ is unavoidable. Let this expression denote the pair of minimally input completed Petri nets needed to compose two processes with the given input alphabets and Petri net models. A definition is straightforward based on the discussion above.

$$(N, I) \bowtie (N', I') = (\lambda\nu. ((N \triangleleft I \cap \nu N'), (N' \triangleleft I' \cap \nu N))) \lambda(P, T, A, M, F). T \quad (5.20)$$

5.4 Process combinators

The process combinators introduced informally in [Section 3.6](#) are now ready to be specified formally in this section with the help of the Petri net operators defined in [Section 5.3](#). For some specifications, the notation $\widetilde{\text{seq}}$, $\widetilde{\text{par}}$, etc. emphasizes their restriction to open Petri net modeled processes in $\tilde{\mathbb{D}}$. Removal of this restriction concludes in [Chapter 7](#).

5.4.1 Communication

The combinators pertaining to communication between a process and its environment are the simplest to specify. Fixing distinct vertices v_i arbitrarily as $\mathbb{V}^{\circ-1} i$ for i ranging from 0 to 3, we may define the combinators $\text{get} : \mathbb{T} \rightarrow \tilde{\mathbb{D}}$ and $\text{put} : \mathbb{T} \rightarrow \tilde{\mathbb{D}}$ as

$$\begin{aligned} \text{get}(a) &= (\{a\}, \emptyset, (\{v_0, v_1, v_2\}, \{a, v_3\}, \{(a, v_1), (v_0, v_3), (v_1, v_3), (v_3, v_2)\}, \{v_0\}, \{v_2\})) \\ \text{put}(b) &= (\emptyset, \{b\}, (\{v_0, v_1, v_2\}, \{b, v_3\}, \{(v_1, b), (v_0, v_3), (v_3, v_1), (v_3, v_2)\}, \{v_0\}, \{v_2\})) \end{aligned}$$

following [Figure 3.10](#). In other words, $\text{get } a$ is a process $(I, O, N) \in \tilde{\mathbb{D}}$ satisfying $I = \{a\}$, $O = \emptyset$, and $N = (P, T, A, M, F) \in \tilde{\mathbb{P}}$ with $P = \{v_0, v_1, v_2\} \in \mathcal{P}(\mathbb{V})$, and so on.

5.4.2 Parallel composition

Parallel composition captures the effect of putting two processes together such that they run concurrently and interact with their environment as noted previously. That is, an input signal belonging to both input alphabets can go to one process or the other, signals belonging to the input alphabet of one process but the output of the other pass unobserved between them, and all other output signals pass independently to the environment.

In keeping with this interpretation, the input alphabet of the parallel composition contains all input symbols from either process except those that are also output symbols, and the output alphabet is similar. For two processes (I, O, N) and (I', O', N') in $\tilde{\mathbb{D}}$, the parallel composition therefore has the input alphabet $(I \cup I') - (O \cup O')$ and the output alphabet $(O \cup O') - (I \cup I')$, expressible more succinctly as $i - o$ and $o - i$ respectively in terms of $i = I \cup I'$ and $o = O \cup O'$.

The Petri net model of each process requires completion with respect to its own input alphabet and the input transitions visible in that of other process. The Petri net model of the parallel composition of the two is therefore $x \uplus y$ for Petri nets $(x, y) = (N, I) \bowtie (N', I')$ by Equation 5.20, implying a whole process

$$\widetilde{\text{par}}((I, O, N), (I', O', N')) = (\lambda(i, o, (x, y)). (i - o, o - i, x \uplus y)) (I \cup I', O \cup O', (N, I) \bowtie (N', I')).$$

5.4.3 Environmental restriction

So much for the one-liners, next we seek a definition for $\widetilde{\text{env}} : \tilde{\mathbb{D}} \times \tilde{\mathbb{D}} \rightarrow \mathbb{D}$ to take a process $X \in \tilde{\mathbb{D}}$ and an environment $E \in \tilde{\mathbb{D}}$ to a process $\text{env}(X, E) \in \mathbb{D}$ capable of exhibiting only the features of X exercised by interacting with E . For X of the form (I, O, N) , this operation need only be defined for values of E of the form (O, I, N') , meaning that the input alphabet of X coincides with the output alphabet of E and *vice versa*.

The result could be given almost immediately by $(I, O, N \uplus N')$ were it not for the need to attend to input and output completion of the Petri net models N and N' . If a set $i \subset \mathbb{T}$ contains output transitions of N' that are not input transitions of N , then something like $N \triangleleft i$ is needed in place of N . If in addition a set $o \subset \mathbb{T}$ contains input transitions from N' that are not output transitions of N , then something like $N \triangleleft i \triangleright o$ is more appropriate. Even if nothing needs to be done to N , analogous transformations to N' could be necessary.

Narrowing down the sets $i, o \subset \mathbb{T}$ mentioned above is easy enough by observing that $i = T \cap I$ and $o = T \cap (\mathbb{T} - I)$ should hold for the input alphabet I and the set of transitions T in the Petri net of the separated sum $N \uplus N' = (P, T, A, M, F)$. By these criteria, i and o also include transitions already present in N , but these have no effect on the input and output completion $N \triangleleft i \triangleright o$.

Expressing this result as $e(N \uplus N') (I, N)$ in terms of a function $e : \mathbb{P} \rightarrow ((\mathcal{P}(\mathbb{T}) \times \tilde{\mathbb{P}}) \rightarrow \tilde{\mathbb{P}})$ defined by

$$e = \lambda(P, T, A, M, F). \lambda(I, N). (\lambda(i, o). N \triangleleft i \triangleright o) (T \cap I, T \cap (\mathbb{T} - I))$$

yields the analogous transformation to N' for free as $e(N \uplus N') (O, N')$, and the overall definition for the $\widetilde{\text{env}}$ combinator as follows.

$$\widetilde{\text{env}}((I, O, N), (O, I, N')) = (I, O, (\lambda f. (f(I, N) \uplus f(O, N')) e(N \uplus N'))) \quad (5.21)$$

5.4.4 Sequential composition

The sequential composition of two processes should have the effect of starting one process first and then the other when the first process attains its designated final marking. The same conventions about alphabets and input completion explained in Section 5.4.2 apply to sequential composition as to parallel composition.

Referring again to Figure 3.10, we note first that sequential composition requires the creation of a new anonymous transition not present in either of the given Petri nets. For open Petri net modeled processes (I, O, N) and (I', O', N') , the pair of Petri nets $(x, y) = (N, I) \bowtie (N', I')$ after mutual input

completion by Equation 5.20 determines a unit set of vertices $t = s_0(x, y) \in \mathcal{P}(\mathbb{V})$ not present in either in terms of the function $s_0 : \mathbb{P} \times \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V})$ given by

$$s_0 = \lambda(x, y). (\lambda t. (\mu ((x \uplus y) \wr (\emptyset, t, \emptyset, \emptyset, \emptyset))) t) \{\min \mathbb{V}\} \quad (5.22)$$

based on Equation 5.15 and Equation 5.16.

Normally there should be exactly one new anonymous transition created, but if the final marking of x or the initial marking of y were empty, then at least one of the preset or the postset of the new transition would be empty in the sequential composition, which would violate one of the conditions for a well formed Petri net stipulated on page 91. To allow for this possibility, let the set of created transitions $t = s_1(x, y) s_0(x, y)$ be given by a function $s_1 : \mathbb{P} \times \mathbb{P} \rightarrow (\mathcal{P}(\mathbb{V}) \rightarrow \mathcal{P}(\mathbb{V}))$ defined as

$$s_1 = \lambda((P, T, A, M, F), (P', T', A', M', F')). \lambda t. (\lambda i. \langle t, \emptyset \rangle_i) \delta_0^{|F||M'|}$$

which coincides with $s_0(x, y)$ if both markings are non-empty but is empty otherwise.

The next thing needed is a set of arcs connecting the finally marked places of the first Petri net x to the new transition in t and the new transition to the initially marked places of the second Petri net y . The required set of arcs is expressible as the result $a = s_2((x, y), t) \in \mathcal{P}(\mathbb{V} \times \mathbb{V})$ of the function $s_2 : (\mathbb{P} \times \mathbb{P}) \times \mathcal{P}(\mathbb{V}) \rightarrow \mathcal{P}(\mathbb{V} \times \mathbb{V})$ defined by

$$s_2 = \lambda((x, y), t). (\lambda((P, T, A, M, F), (P', T', A', M', F')). (F \times t) \cup (t \times (\mu (x \wr y)) M')) (x, y)$$

where the separation function $x \wr y$ is mapped over M' in anticipation of adding this set of arcs to the coalesced sum of x and y . Note that $s_2((x, y), t)$ is empty whenever t is empty.

A Petri net $s_3(x, y) \in \mathbb{P}$ including both the new transition set t and the new arcs a with respect to the Petri nets x and y is conveniently summarized by a function $s_3 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ defined as follows.

$$s_3 = \lambda z. (\lambda t. (\lambda a. (\emptyset, t, a, \emptyset, \emptyset)) s_2(z, t)) (s_1 z) (s_0 z)$$

The overall Petri net model for the sequential composition is almost $(x \cup y) \uplus s_3(x, y)$, that is, the coalesced sum of the originals with the extra transition and arcs given by $s_3(x, y)$ thrown in, except that its set of initially marked places should be limited to those of x , and its set of finally marked places should be limited to those of y . If we replace x in the coalesced sum with a Petri net $x \dot{-} (\emptyset, \emptyset, \emptyset, \emptyset, \mathbb{V})$, which is similar to x by Equation 5.18 but has an empty final marking, and replace y with a Petri net $y \dot{-} (\emptyset, \emptyset, \emptyset, \mathbb{V}, \emptyset)$, which is similar to y but has an empty initial marking, then the result is as it should be. A Petri net $s_4(x, y) \in \mathbb{P}$ incorporating this last adjustment is expressible in terms of a function $s_4 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ defined as follows.

$$s_4 = \lambda(x, y). ((x \dot{-} (\emptyset, \emptyset, \emptyset, \emptyset, \mathbb{V})) \cup (y \dot{-} (\emptyset, \emptyset, \emptyset, \mathbb{V}, \emptyset))) \uplus s_3(x, y)$$

The rest of the definition of the sequential composition combinator $\widetilde{\text{seq}} : \widetilde{\mathbb{D}} \times \widetilde{\mathbb{D}} \rightarrow \widetilde{\mathbb{D}}$ deals only with the conventions regarding input and output alphabets mentioned above.

$$\widetilde{\text{seq}}((I, O, N), (I', O', N')) = (\lambda(i, o). (i - o, o - i, s_4((N, I) \bowtie (N', I')))) (I \cup I', O \cup O') \quad (5.23)$$

That is, the input alphabet overall is the union of the given input alphabets excluding the outputs, and the output alphabet is constructed similarly.

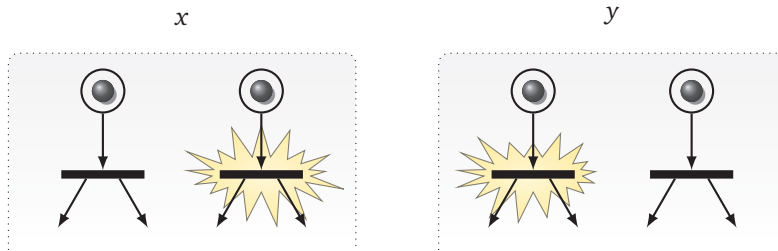


Figure 5.7: Two Petri nets x and y each with two initial places need to be combined by $\widetilde{\text{alt}}$ with mutual exclusion, not with transitions allowed to fire concurrently in x and y as shown here.

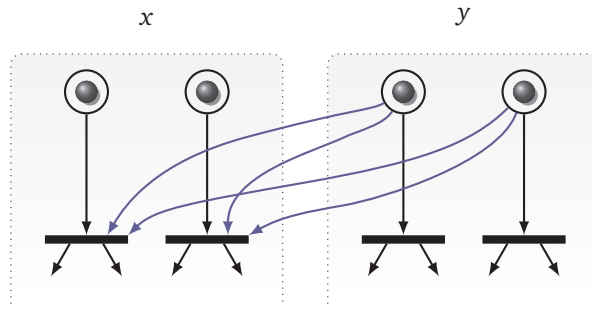


Figure 5.8: An arc from every initially marked place in y to every initially enabled transition in x disables y when x starts executing.

5.4.5 Choice

The definition of the $\widetilde{\text{alt}}$ combinator in this section unfortunately continues the trend toward increasing difficulty. For two open Petri net modeled DI processes (I, O, N) and (I', O', N') , the process

$$\widetilde{\text{alt}}((I, O, N), (I', O', N')) \in \widetilde{\mathbb{D}}$$

should interact with the environment as if exactly one process or the other executes in a mutually exclusive and atomic (*i.e.*, all-or-nothing) way. Similar conventions regarding alphabets and input completion to parallel and sequential combination are also relevant. A brief overview of the difficulties involved in initialization and termination follows before the formal specification.

Initialization

When Petri nets $x, y \in \widetilde{\mathbb{P}}$ are combined due to the $\widetilde{\text{alt}}$ combinator, one aspect of the behavior to be avoided is shown in Figure 5.7. If one of them starts executing due to one of its initially enabled transitions firing, the other one must be prevented from doing so. One way of constructing a combined Petri net to address this issue is to form the separated sum of x and y , and then to insert extra arcs as needed to enforce the required mutual exclusion.

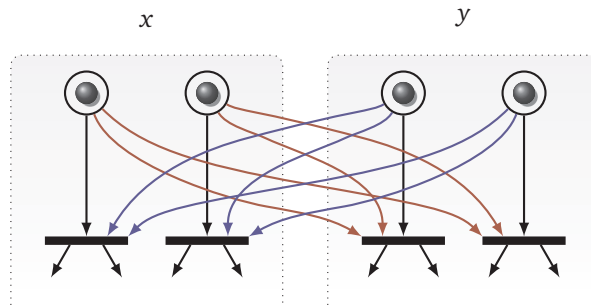


Figure 5.9: Arcs in both directions let them disable each other.

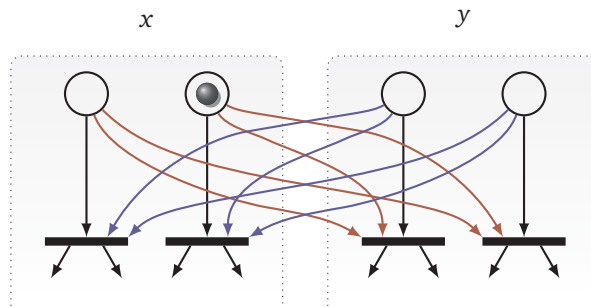


Figure 5.10: Unfortunately, after the left transition in x fires, not only is y disabled, but also the other transition in x .

For a first attempt along these lines, let two Petri nets x and y have initially marked places and initially enabled transitions as shown in Figure 5.8. For the sake of this discussion, let the initially marked places in each operand, their postset transitions, and the arcs connecting them be described informally as the operand's *initialization network*. If the initially marked places in y are connected by arcs (shown in blue) to the transitions in the initialization network of x , then the onset of any activity in x immediately evacuates them. With its initially marked places emptied, y can not execute, so mutual exclusion is ensured. Alternatively, if y starts executing first, then x should be similarly disabled. Hence, the arcs shown in red in Figure 5.9 should be created to connect the initially marked places x to transitions in the initialization network of y .

This solution would be perfect were it not for the disadvantage illustrated in Figure 5.10. The firing of the left transition in x has caused the evacuation of the initially marked places in y , as intended. However, these vacant places now have an inhibitory effect on the remaining transition in x due to the arcs created to evacuate them if it had fired first. Although mutual exclusion is certain, deadlock is inevitable as well. A similar argument applies of course to any transition in either initialization network.

To solve this problem, we have to do something like what is shown in Figure 5.11, which is to keep the arcs between the operands x and y as in the first attempt above, but to create a copy of the initialization network in each operand. The result is an initialization network in two stages for each

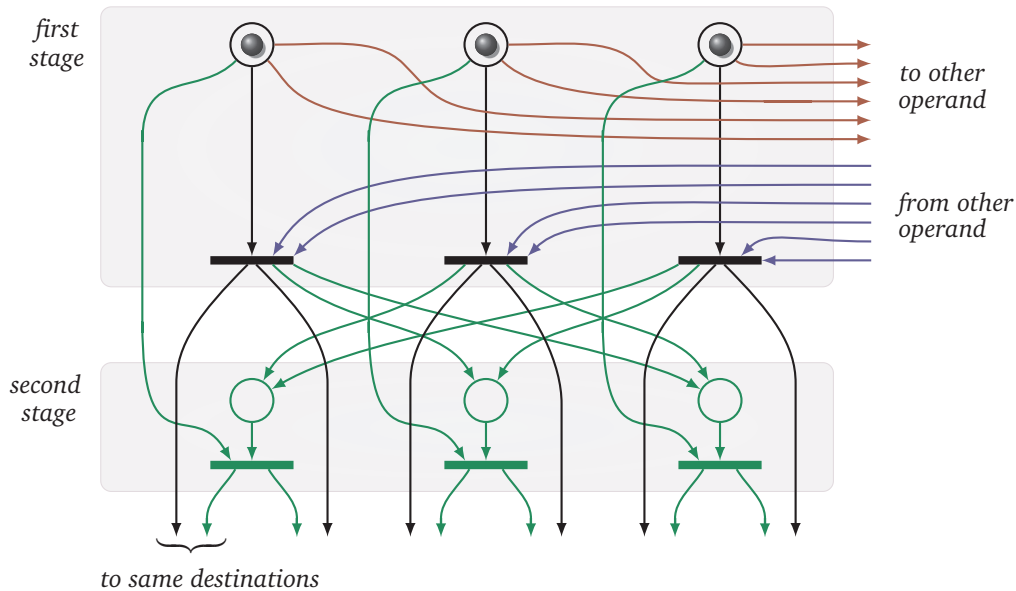


Figure 5.11: Making a copy of the initialization network of an $\tilde{\text{alt}}$ combinator operand in a second stage lets it block the other operand (not shown) without deadlocking itself.

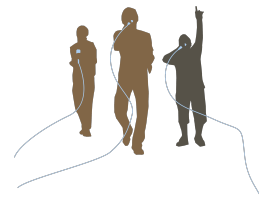
operand. The first stage contains the original initialization network, and the second stage contains a substitute for each place and each transition in the first stage, with no connections between the second stage and the other operand.

The two-stage initialization network avoids deadlock by creating a second execution path. When any transition in the first stage fires, it marks every place in the second stage other than those substituting for members of its own preset. Although the other operand indirectly disables the other transitions in the first stage, the transition that fires in the first stage enables their substitutes in the second stage. Each of the second stage transitions has the same postset as the one for which it substitutes in the first stage, hence the same effect when it fires. It also empties the corresponding initially marked places in the first stage.

Termination

As well as initializing the combined Petri net correctly, terminating it is also important. The operand that executes indicates termination of the combination when it terminates. In general each operand may have more than one finally marked place, so a construction like the one shown in Figure 5.12 must coordinate them. For each Petri net model x and y , a new transition synchronizes its finally marked places. These transitions must feed into a newly created finally marked place, which receives a token therefore whenever either of them fires. The marking of this place signifies termination of the combination.

It would be unsafe for both transitions in the termination network to fire concurrently, because



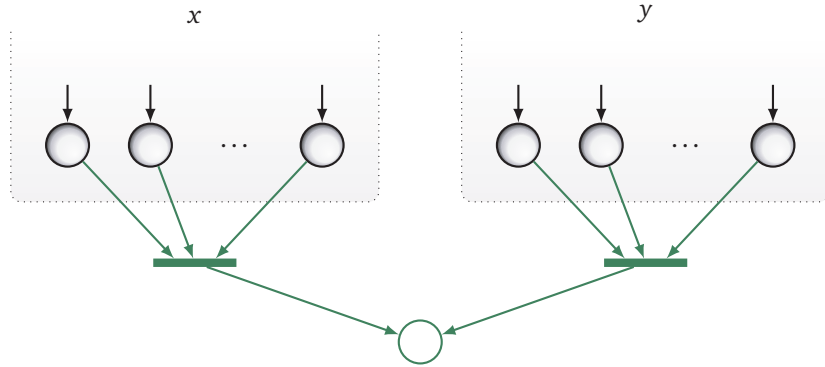


Figure 5.12: general form of the termination network for a pair of Petri nets x and y with multiple final places when combined by the $\widetilde{\text{alt}}$ combinator (new additions in green)

then the new final place would receive two tokens. Only mutually exclusive execution of x and y during initialization prevents this outcome.

Specification

Similarly to sequential composition, the bulk of the formal specification of the $\widetilde{\text{alt}}$ combinator concerns the construction of the Petri net model resulting from the pair of mutually input completed Petri nets $(x, y) = (N, I) \bowtie (N', I')$ determined by the operands (I, O, N) and (I', O', N') . This construction relies heavily on the foregoing overview for context and motivation.

Initialization networks The initialization network is a good place to start. A Petri net $x \in \mathbb{P}$ induces a pair

$$(M, t) = c_0 x \in \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V})$$

of initially marked places and their postset transitions with $c_0 : \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V})$ given by

$$c_0 = \lambda(P, T, A, M, F). (M, \mathcal{R}(A \cap (M \times \mathbb{V}))) \quad (5.24)$$

wherein the right side t determines the whole initialization network $x' = c_1(t, x) \in \mathbb{P}$ for x , including the arcs local to it, with $c_1 : \mathcal{P}(\mathbb{V}) \times \mathbb{P} \rightarrow \mathbb{P}$ given by

$$c_1 = \lambda(t, (P, T, A, M, F)). (M, t, A \cap ((M \times t) \cup (t \times M)), \emptyset, \emptyset).$$

Internal arcs Because the second stage initialization network is a copy of the original, we can expect something of the form $x \uplus x'$ to figure in the result, with arcs added to it as shown in Figure 5.11. Focusing just on the arcs between x and its second stage initialization network x' for the moment, let $f = x \wr x'$ denote the separation function, let t denote the postset transitions of initially marked places in x as above, and let A and M denote arcs and the initial marking of x respectively. Then



for example a transition $a \in t$ is a first stage transition in $x \uplus x'$, but $f a$ is the corresponding second stage transition. The arcs needing to be added to $x \uplus x'$ to make all of the connections shown in the figure fall into three classes:

- arcs from the second stage transitions to the postsets of the first stage transitions

$$\bigcup_{(a,b) \in A \cap (t \times \mathbb{V})} \{(f a, b)\}$$

- arcs from the first stage places to the second stage transitions

$$\bigcup_{(c,d) \in A \cap (\mathbb{V} \times t)} \{(c, f d)\}$$

- and arcs from the first stage transitions to the second stage places.

$$\bigcup_{e \in M - \mathcal{D}(A \cap (\mathbb{V} \times \{i\}))} (\mu \lambda i. \bigcup \{(i, f e)\}) t$$

Note that places c in members (c, d) of $A \cap (\mathbb{V} \times t)$ may include more than just members of M , such as the postset places of any input transitions, which is as it should be. The last expression is more complicated because each first stage transition $i \in t$ connects only to the second stage places $f e$ corresponding to first stage places $e \in M$ that are outside the preset of i .

Let the set of all arcs in any of these three classes be denoted $c_2(x \wr x', t, x)$ in terms of a function

$$c_2 : ((\mathbb{T} \cup \mathbb{V}) \rightarrow (\mathbb{T} \cup \mathbb{V})) \times \mathcal{P}(\mathbb{V}) \times \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V} \times \mathbb{V})$$

defined as follows

$$c_2 = \lambda(f, t, (P, T, A, M, F)). \bigcup_{(a,b) \in A \cap (t \times \mathbb{V})} \{(f a, b)\} \cup \bigcup_{(c,d) \in A \cap (\mathbb{V} \times t)} \{(c, f d)\} \cup \bigcup_{e \in M - \mathcal{D}(A \cap (\mathbb{V} \times \{i\}))} (\mu \lambda i. \bigcup \{(i, f e)\}) t$$

so that the concept of a Petri net x augmented by its second stage initialization network corresponds roughly to $c_3 x$ in terms of a function $c_3 : \mathbb{P} \rightarrow \mathbb{P}$ given by

$$c_3 = \lambda x. (\lambda(M, t). (\lambda x'. (x \uplus x') \uplus (\emptyset, \emptyset, c_2(x \wr x', t, x), \emptyset, \emptyset)) c_1(t, x)) c_0 x. \quad (5.25)$$

Termination networks Passing on to the termination network, we can start by augmenting each Petri net x and y by a set a of arcs connecting each member of its final marking to an arbitrary distinct anonymous transition. For a Petri net $x = (P, T, A, M, F)$ and a set of arcs $a = F \times \{\min \mathbb{V}\}$, this effect is achievable by transforming x to $x \uplus (\emptyset, \mathcal{R}(a), a, \emptyset, \emptyset)$. After this change, there is no further use for what was formerly the final marking F , so x can be transformed further to

$$(x \uplus (\emptyset, \mathcal{R}(a), a, \emptyset, \emptyset)) \div (\emptyset, \emptyset, \emptyset, \emptyset, F)$$

emptying the final marking, or more briefly to $c_4 x$ in terms of a function $c_4 : \mathbb{P} \rightarrow \mathbb{P}$ defined as

$$c_4 = \lambda x. (\lambda(P, T, A, M, F). (\lambda a. (x \uplus (\emptyset, \mathcal{R}(a), a, \emptyset, \emptyset)) \div (\emptyset, \emptyset, \emptyset, \emptyset, F)) (F \times \{\min \mathbb{V}\})) x$$

with $c_4 y$ expressing a similar transformation of the other Petri net y . Petri nets of the form $c_4 x$ or $c_4 y$ are not well formed because the new anonymous transitions added to them have empty postsets, but a remedy is imminent. Note that this transformation has no effect if the final marking F is already empty.

The rest of the termination network consists of a single place distinct from any vertices in the two Petri nets, and an arc connecting each of the newly created transitions to it. Given a Petri net $z = x \cup y$ with x and y both transformed already by c_4 , we express a singleton set p of places disjoint from their vertices and the newly created transitions as $p = c_5 z$ with $c_5 : \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V})$ given by

$$c_5 = \lambda(P, T, A, M, F). \{ \min(\mathbb{V} - (P \cup T)) \}. \quad (5.26)$$

To effect the connections, z can be transformed to $z \uplus (p, \emptyset, t \times p, \emptyset, p)$, where t contains the new anonymous transitions in z , thereby adding an arc from them to the new place in p . Finding t is easy because it contains the only anonymous transitions with empty postsets, making it stand out as $(T \cap \mathbb{V}) - \mathcal{D}(A)$, where T is the set of transitions and A is the set of arcs in z . Let this result be denoted $(c_6 c_5 z) z$ in terms of a function $c_6 : \mathcal{P}(\mathbb{V}) \rightarrow (\mathbb{P} \rightarrow \mathbb{P})$ given by

$$c_6 = \lambda p. \lambda z. (\lambda(P, T, A, M, F). (\lambda t. z \uplus (p, \emptyset, t \times p, \emptyset, p))) ((T \cap \mathbb{V}) - \mathcal{D}(A)) z. \quad (5.27)$$

Inter-operand arcs Having dealt with the initialization and termination networks for the most part, we are still faced with the inter-operand arcs depicted in [Figure 5.11](#). There needs to be an arc from every initially marked place in each operand to every transition in the postset of any initially marked place in the other. Roughly speaking, if $(M, t) = c_0 x$ and $(M', t') = c_0 y$ are the initially marked places and their postsets for each operand by [Equation 5.24](#), then we need arcs from M to t' and from M' to t . However, by the time these arcs are added, x and y will have been combined as a coalesced sum, so some members of M' or t' could be different. Letting $f = \mu(x \wr y)$ denote their mapped separation function, we can allow for this possibility by connecting M to $f t'$ and $f M'$ to t in a set of arcs $c_7(x, y)$, with $c_7 : \mathbb{P} \times \mathbb{P} \rightarrow \mathcal{P}(\mathbb{V} \times \mathbb{V})$ defined as

$$c_7 = \lambda(x, y). (\lambda(f, (M, t), (M', t')). (M \times f t' \cup ((f M') \times t)) (\mu(x \wr y), c_0 x, c_0 y)).$$

Synthesis To start wrapping things up, we can add the set of arcs $c_7(x, y)$ to coalesced sum $x \cup y$

$$z = (x \cup y) \uplus (\emptyset, \emptyset, c_7(x, y), \emptyset, \emptyset)$$

temporarily denoting that result z , and then add the termination network as discussed above to a result $c_8(x, y)$ with $c_8 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ given by

$$c_8 = \lambda(x, y). (\lambda z. (c_6 c_5 z) z) ((x \cup y) \uplus (\emptyset, \emptyset, c_7(x, y), \emptyset, \emptyset)).$$

For this function to be useful, the arguments x and y should have their second stage initialization networks already incorporated by c_3 ([Equation 5.25](#)) and their termination network transition already incorporated by c_4 , but rather than remembering all that, let $c_9 : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ take Petri nets x and y in their original forms to the same result $c_9(x, y)$ by

$$c_9 = \lambda(x, y). c_8(c_4 c_3 x, c_4 c_3 y).$$

With $c_9((N, I) \bowtie (N', I'))$ accounting for the Petri net model, the rest of the $\widetilde{\text{alt}}$ combinator definition is similar to that of sequential composition, with unions of the corresponding alphabets and the internal signals hidden.

$$\widetilde{\text{alt}}((I, O, N), (I', O', N')) = (\lambda(i, o). (i - o, o - i, c_9((N, I) \bowtie (N', I')))) (I \cup I', O \cup O')$$

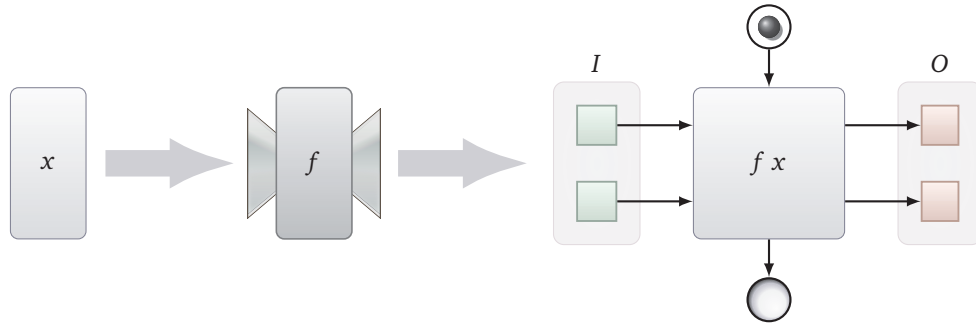


Figure 5.13: Feeding an argument x with empty alphabets to a function f generates a result fx with possibly non-empty alphabets I and O .

5.4.6 Recursion

One more process combinator completes the set of those necessary for describing interesting and useful behavior, especially when repetition is required. A function



$$\mathbf{fix} : (\mathbb{D} \rightarrow \mathbb{D}) \rightarrow \mathbb{D}$$

takes a function $f : \mathbb{D} \rightarrow \mathbb{D}$ to a process $\mathbf{fix} f \in \mathbb{D}$ satisfying $\mathbf{fix} f \equiv f(\mathbf{fix} f)$ by a definition of behavioral equivalence to be made precise in Chapter 7. The function \mathbf{fix} is called a pseudo-fixed point combinator instead of a fixed point combinator because the latter would imply equality between $\mathbf{fix} f$ and $f(\mathbf{fix} f)$. Behavioral equivalence is a weaker condition than equality but adequate for our purposes.

In any case, a function that behaves like a fixed point combinator is a general and flexible primitive for various patterns of flow and control. For the simplest example, an expression of the form $\mathbf{fix} \lambda x. \mathbf{seq}(p, x)$ for some constant process $p \in \mathbb{D}$ is equivalent to p repeated infinitely many times. See Section 3.6.2 for further discussion and motivation.

The rest of this section contains an intuitive overview of how one might go about defining a pseudo-fixed point combinator, followed by a fairly short formal specification, followed by a few examples exploring its implications.

Overview

Unlike other process combinators, \mathbf{fix} does not take a process or a signal as an operand, but a function f , so there is no opportunity to inspect or deconstruct the operand in any way. The only way to gather information about f is by applying it to selected inputs and observing the results. For example, applying f to a block of wood $x = (\emptyset, \emptyset, (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset))$, we obtain

$$f(\emptyset, \emptyset, (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)) = (I, O, N) \tag{5.28}$$

with some alphabets I and O and some possibly not very useful Petri net N . This operation is depicted in Figure 5.13.

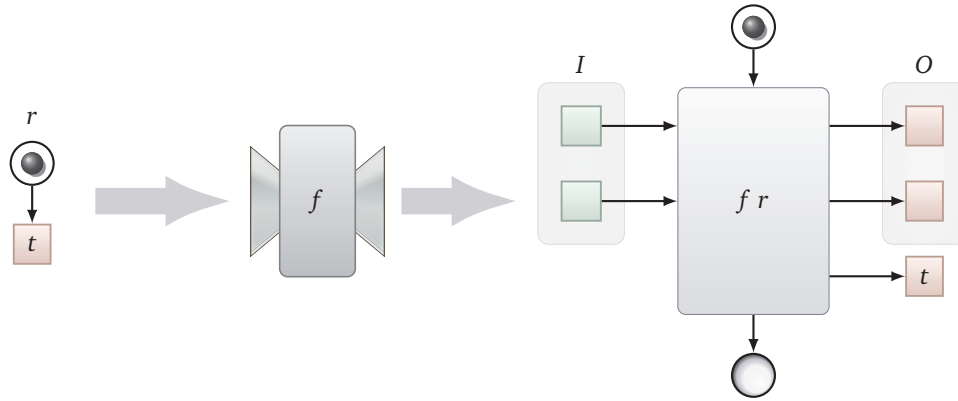


Figure 5.14: Feeding f an operand r with an output alphabet $\{t\}$ disjoint from the alphabets I and O in Figure 5.13 generates a result $f r$ with the output t visible.

Nevertheless, this operation at least allows particular input and output alphabets to be associated with f . If f is defined by process combinators, then the alphabets of $f(I', O', N')$ for an arbitrary operand $(I', O', N') \in \mathbb{D}$ can not vary unpredictably. By a routine inductive argument, they may contain symbols only from I' , O' , I , and O , the latter two being given by Equation 5.28.

This insight suggests that a reference operand r with an empty input alphabet and an output alphabet $\{t\}$ for some t specially chosen to be a non-member of $I \cup O$ by Equation 5.28 would yield a result $f r$ with an input alphabet I and an output alphabet $O \cup \{t\}$ for any f expressible by process combinators. This operation is depicted in Figure 5.14.

Interesting though it may be, this exercise seems to bring us no closer to defining the pseudo-fixed point combinator, so a more radical approach is in order. Generally f takes an operand x having some Petri net model associated with it and builds some bigger and better DI process from it with another Petri net model. If we search carefully through the Petri net model of $f x$ and ignore all the ways this plan could go wrong, we might find one or more copies of that of x in it as shown in Figure 5.15. From this point, it is not much more of a leap to reason as follows.

- If the sought-after pseudo-fixed point x satisfying $x \equiv f x$ were known in advance by magic, it could be plugged into f like any other operand, and its image would occupy the same positions in the resulting Petri net model of $f x$ as the image of any operand normally would.
- Moreover, if the flow of control (manifested in the progress of Petri net tokens) were to reach one of these x images within $f x$, the externally observable behavior thereafter would be the same as if the whole process $f x$ had restarted, because x is behaviorally equivalent to $f x$ by hypothesis.
- It would be tempting therefore to optimize the Petri net model of $f x$ by connecting the initial transitions in each x image directly to the initial places of $f x$ instead of leaving them connected to the rest of the x image, because doing so would have no observable consequences inasmuch as x is behaviorally equivalent to $f x$.
- With this optimization, the rest of the x images would be cut off from any reachable execution path and could be deleted, also without consequences, as shown in Figure 5.16.

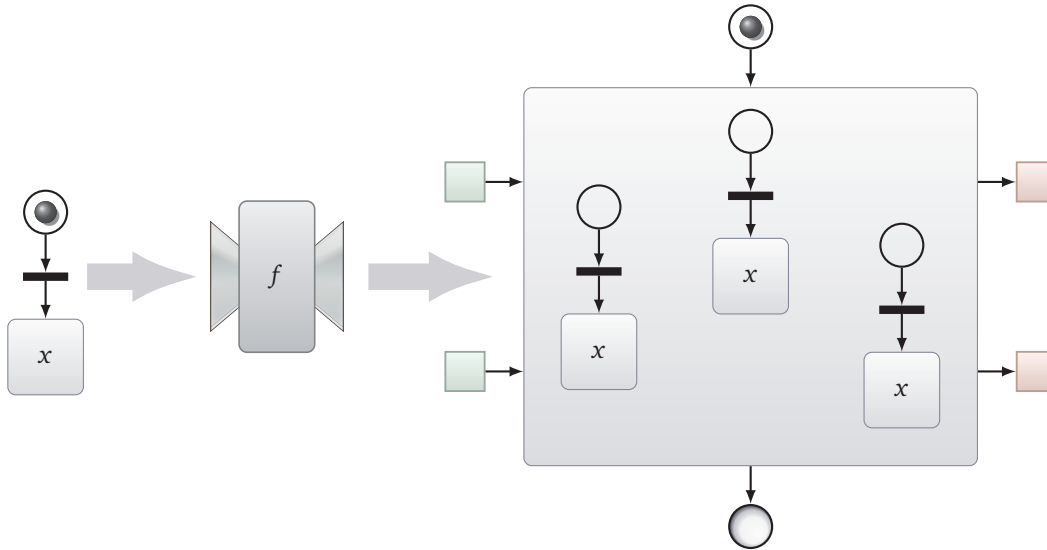


Figure 5.15: Plugging an operand x into f can create a Petri net model $f x$ with copies of x embedded in it.

- Because the rest of each x image is either absent or never reached as a result of this optimization, any operand with an initial place connected to an initial transition could have been supplied for x with the same effect.

This line of reasoning could lead to a formal specification for the pseudo-fixed point combinator were it not for two obstacles. One is the lack of an efficient and reliable procedure for locating the images of the operand x in the resulting model $f x$. The other is that these images might not even be present if the Petri net has been transformed extensively along the way. Such transformations may be necessary if the operand or any intermediate result is a closed Petri net in need of conversion to an open form as described in [Chapter 7](#).

There is nevertheless something to be gathered from this investigation. Computing the pseudo-fixed point of a function can indeed start by applying the function to some operand x . Contrary to the impossible requirement of a solution known in advance, an arbitrary reference input suffices. Furthermore, a reference input whose execution manifests an observable signal would be preserved by any semantically valid Petri net transformation, thereby overcoming the remaining obstacles. For any function f built from process combinators, such an input r is easily obtained as mentioned previously. Proceeding as above, then anonymizing the signal transition t due to r and finally connecting it to the initially marked places would yield the pseudo-fixed point as a result. This operation is depicted in [Figure 5.17](#).

Specification

The formal specification of the pseudo-fixed point combinator presupposes a function $\gamma : \mathcal{P}(\mathbb{T}) \rightarrow \mathbb{T}$ taking any finite subset $s \subset \mathbb{T}$ to a non-member of its operand $\gamma s \notin s$ as discussed in [Section 5.2.3](#). The output alphabet symbol $t = r_0 f \in \mathbb{T}$ needed for the reference input to f then follows from a

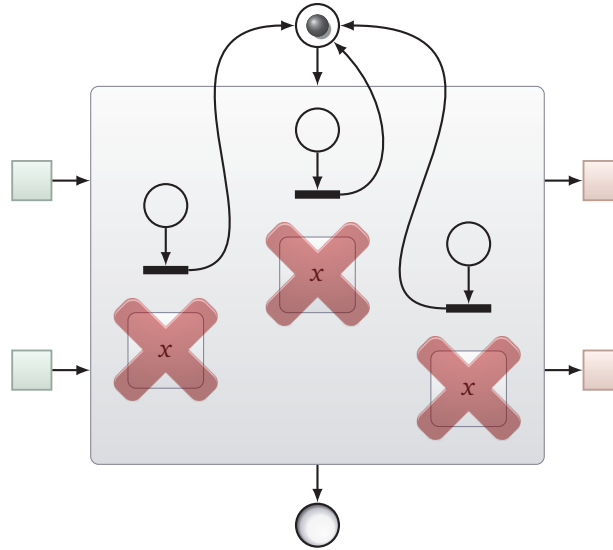


Figure 5.16: If x were a pseudo-fixed point of f , it would make no observable difference to connect the initial transitions of the copies of x to the initial places of $f x$ and delete the rest.

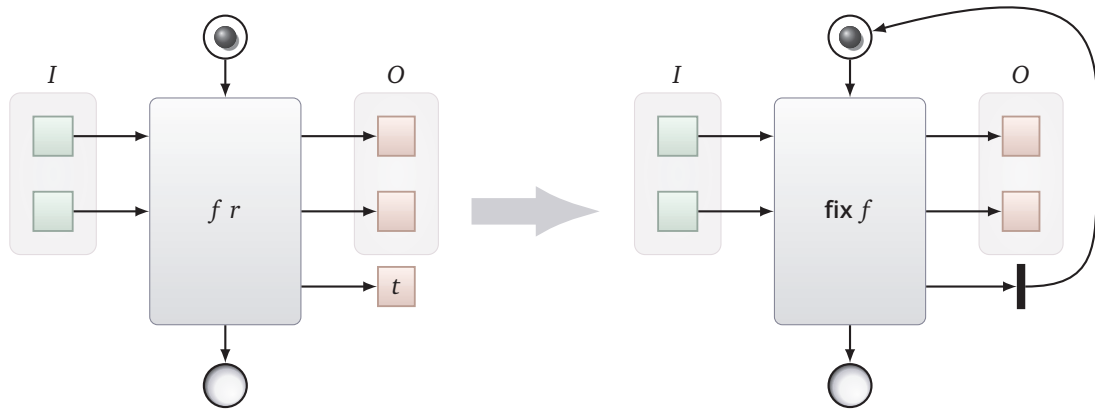


Figure 5.17: Anonymizing and connecting the reference output on $f r$ to the initially marked places yields the pseudo-fixed point of f (cf. Figure 5.14).

function $r_0 : (\mathbb{D} \rightarrow \mathbb{D}) \rightarrow \mathbb{T}$ given by

$$r_0 = \lambda f. (\lambda(I, O, N). \gamma(I \cup O)) f(\emptyset, \emptyset, (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)) \quad (5.29)$$

and the reference input itself from $r_1 t$ with $r_1 : \mathbb{T} \rightarrow \mathbb{D}$ given by

$$r_1 = \lambda t. (\lambda p. (\emptyset, \{t\}, (\{p\}, \{t\}, \{(p, t)\}, \{p\}, \emptyset))) \min \mathbb{V} \quad (5.30)$$

as shown in Figure 5.14. Based on everything up to this point, the next logical step would be to define **fix** f as

$$(\lambda t. (\lambda(I, O, N). (I, O, (r_2 t) N)) f r_1 t) r_0 f$$

where $r_2 : \mathbb{T} \rightarrow (\mathbb{P} \rightarrow \mathbb{P})$ takes the signal t to a function $r_2 t : \mathbb{P} \rightarrow \mathbb{P}$ capturing the transformation depicted in Figure 5.17 by a definition

$$r_2 = \lambda t. \lambda(P, T, A, M, F). (P, T, \{t\} \times M, M, F) \setminus \{t\}$$

using the anonymization operator given in Equation 5.14. However, for reasons to be justified shortly, it is advantageous to create an additional marked place to accompany those already in M if they all happen to land within the preset of t . (Essentially this feature renders t unsafe and declares the resulting process divergent when the condition holds.) To this end, the preset of t expressed as

$$p = \mathcal{D}(A \cap (\mathbb{V} \times \{t\}))$$

and the condition captured by $i = \delta_M^{p \cap M}$ make the expression

$$m = \langle \emptyset, \{\min \mathbb{V}\} \rangle_i \in \mathcal{P}(\mathbb{V})$$

accordingly either empty or a unit set. The Petri net $(m, \emptyset, \emptyset, m, \emptyset)$ given by $(r_3 t) N$ with the function $r_3 : \mathbb{T} \rightarrow (\mathbb{P} \rightarrow \mathbb{P})$ defined as

$$r_3 = \lambda t. \lambda(P, T, A, M, F). (\lambda m. (m, \emptyset, \emptyset, m, \emptyset)) (\lambda i. \langle \emptyset, \{\min \mathbb{V}\} \rangle_i) (\lambda p. \delta_M^{p \cap M}) \mathcal{D}(A \cap (\mathbb{V} \times \{t\}))$$

allows for the substitution of $N \uplus (r_3 t) N$ for N in the pseudo-fixed point definition originally proposed above, with the actual definition as follows.

$$\mathbf{fix} f = (\lambda t. (\lambda(I, O, N). (I, O, ((r_2 t) (N \uplus (r_3 t) N)))) f r_1 t) r_0 f \quad (5.31)$$

Examples

The rest of this section concerns several examples of the **fix** combinator, the first being somewhat useful, and the others involving divergent processes constructed to investigate certain edge cases. The useful one is about using the **fix** combinator to solve

$$X \equiv \widetilde{\mathbf{seq}}(C, X)$$

for $X \in \mathbb{D}$ in terms of a known constant process $C \in \mathbb{D}$. One way of reading this equation is that X is equivalent to a process that starts by behaving like C and then proceeds to behave like itself. Another reading is that performing C first and then X is the same as just performing X . In either case, a process that repeats C *ad infinitum* would fill the bill.

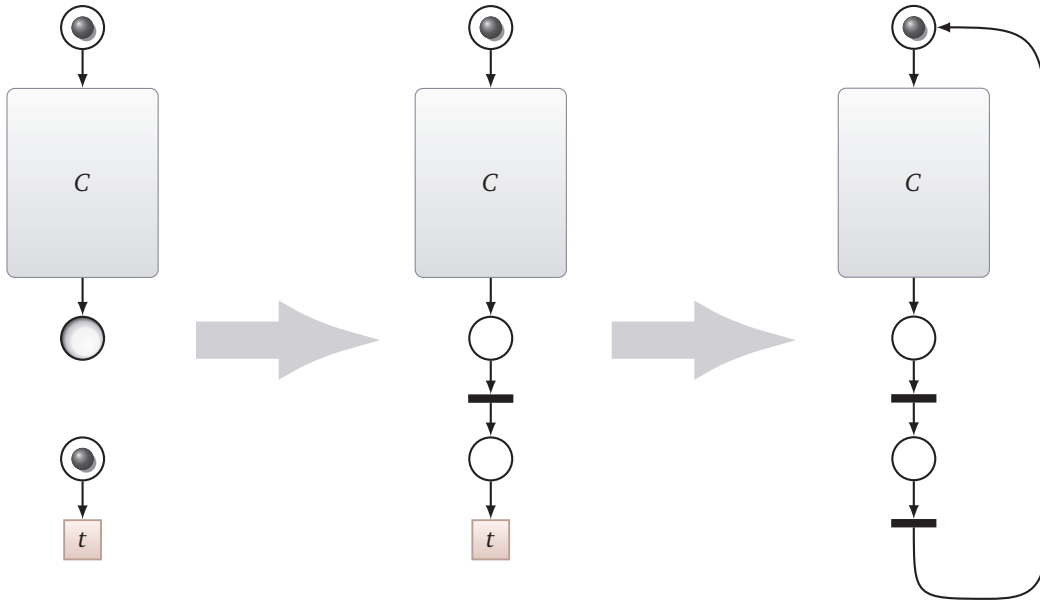


Figure 5.18: Evaluation of $\text{fix } \lambda x. \widetilde{\text{seq}}(C, x)$ starts with C and the reference input, left, combined by sequential composition, center, followed by the rerouting of an arc to the initial place, right.

A solution given by the **fix** combinator as defined above is of course

$$X = \text{fix } \lambda x. \widetilde{\text{seq}}(C, x).$$

To spell out the computation in greater detail than usual, we are solving for X by applying the operand of the **fix** combinator, $\lambda x. \widetilde{\text{seq}}(C, x)$, to the reference argument

$$r_1 r_0 \lambda x. \widetilde{\text{seq}}(C, x)$$

per Equation 5.29 and Equation 5.30 using the usual definition of sequential composition by Equation 5.23 in

$$(\lambda x. \widetilde{\text{seq}}(C, x)) r_1 r_0 \lambda x. \widetilde{\text{seq}}(C, x) = \widetilde{\text{seq}}(C, r_1 r_0 \lambda x. \widetilde{\text{seq}}(C, x)).$$

This computation entails connecting the final place of C to the initial place in the reference input as shown in Figure 5.18. The anonymization of t and its connection to the initial place of C completes the construction.

Does this solution accord with intuition? The result shown at the right of Figure 5.18 is notable for not having a final marking. If it were sequentially composed with another process, the latter would never get a chance to execute. However, if C is a process that always terminates, then X is a process that repeats C forever.

A less useful but more instructive example is easily found in the case of

$$X \equiv \widetilde{\text{seq}}(X, C).$$

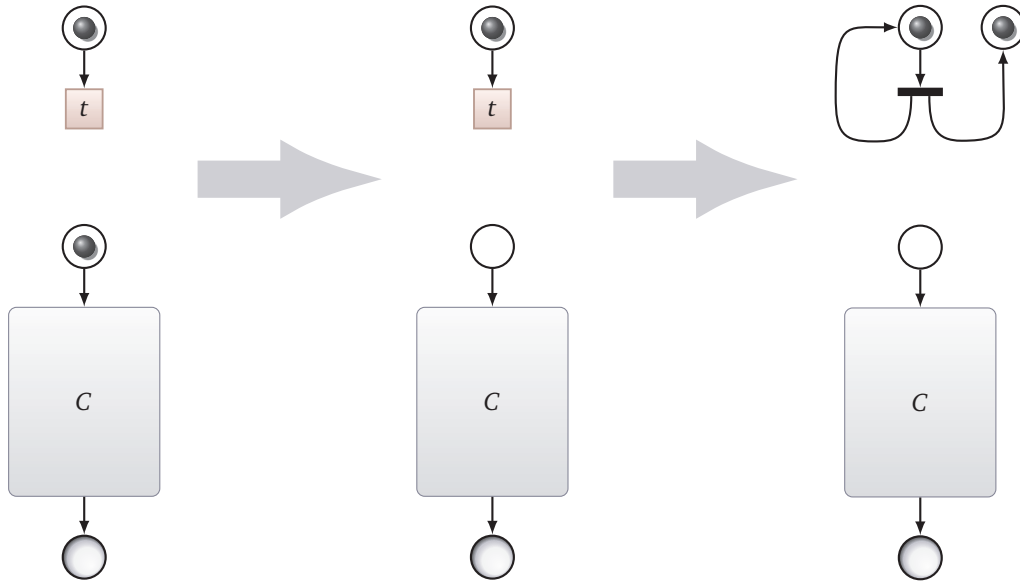


Figure 5.19: For $\mathbf{fix} \lambda x. \widetilde{\mathbf{seq}}(x, C)$ a specially created place ensures a divergent process model.

Proceeding as above, we could read X as a process that starts by behaving like itself and then like C , or as a process for which performing C afterwards would make no observable difference. Despite superficial similarities, this behavior is much less constrained than the previous example. A process X that repeats C indefinitely fits this description, but so does one that deadlocks immediately, one that engages in some arbitrary interaction and then deadlocks, and even one that behaves completely unpredictably at all times. With multiple solutions available, we are obliged to decide on one.

The decision implicit in the \mathbf{fix} combinator definition for a case like this one is a matter of routine calculation by evaluating

$$X = \mathbf{fix} \lambda x. \widetilde{\mathbf{seq}}(x, C).$$

Applying $\lambda x. \widetilde{\mathbf{seq}}(x, C)$ to a reference operand with an output alphabet $\{t\}$ results in a process with a Petri net model N whose only initially marked place is in the preset of t . This condition calls for an additional marked place in $N \uplus (r_3 \ t) \ N$ leading to the result shown in [Figure 5.19](#)

The marked place in the postset of the initially enabled transition in the Petri net shown at the right of [Figure 5.19](#) means the Petri net is unsafe. Any process modeled by an initially unsafe Petri net is divergent and by convention completely unpredictable. It may be tempting to make stronger inferences about this particular process, for example due to the lower part C being disabled and therefore unable to output, but any such inference would be mistaken because the semantics implied by the work of [Chapter 6](#) and [Chapter 7](#) admits no gradations in divergence.

Hence we have our answer about the solution indicated by definition of the \mathbf{fix} combinator, but is this solution ideal? Anything more useful than a completely unpredictable process would seem preferable, such as an infinite repetition of the constant C , but oddly enough, there are good reasons to think the worst solution is actually the best:

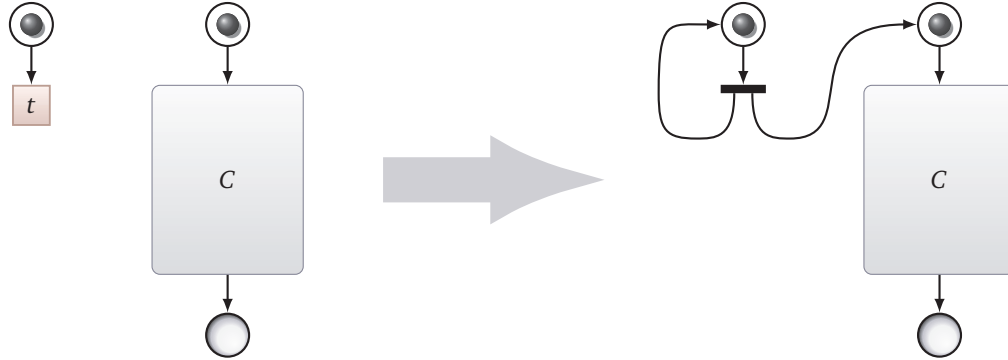


Figure 5.20: evaluation of $\text{fix } \lambda x. \widetilde{\text{par}}(x, C)$, which is the same as $\text{fix } \lambda x. \widetilde{\text{par}}(C, x)$

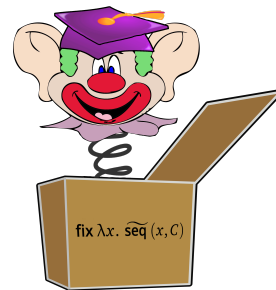
- Process combinator expressions are intended not just to be translated into useful circuits, but to define the environments in which circuits are required to operate. The most cautious assumption about an environment is the least constrained.
- Even in the case of process combinators being used for writing circuit specifications, the divergent interpretation helps to avoid any misunderstanding because the most expressive specification clearly identifies the least capable circuit meeting it.
- Sequential composition is not generally commutative, so nothing is achieved aesthetically by forcing it to be so in this case (*i.e.*, by equating $\text{fix } \lambda x. \widetilde{\text{seq}}(x, C)$ to $\text{fix } \lambda x. \widetilde{\text{seq}}(C, x)$). If any mathematical theory were relevant, it would be that of complete partial orderings, where least fixed points are the most natural solutions. (See [Appendix B](#).)

In summary, for an equation that does not fully constrain its solution, the best solution is the least specific one that satisfies it. The least specific process of all is one that may behave unpredictably. This process is associated with any Petri net model possessed of an initially unsafe marking. The fix combinator is defined to produce such a model when appropriate.

This observation suggests a rule of thumb whereby a divergent process should be the first solution attempted for any equation, with no need to look further if it works. For example, the equation

$$X \equiv \widetilde{\text{par}}(X, C)$$

where C is some constant process, has a divergent process as a solution because if X interacts unpredictably with its environment, then so does X running concurrently with something else. This conclusion holds despite the divergent solution being in some sense a less natural choice here than in the previous example. An intuitive way of thinking about the solution is as the limit of C composed with itself infinitely many times in parallel. For many useful instances of C , parallel composition is idempotent (up to behavioral equivalence) and this limit is arguably C , but



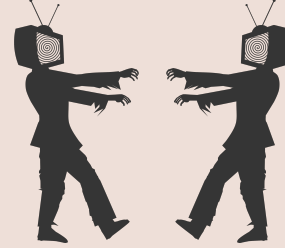
this property evidently does not extend to non-quiescent processes (e.g., $C = \mathbf{put} \ b$), whereas the divergent solution covers everything. It is reassuring to note that the solution obtained by the **fix** combinator is the divergent one as shown in [Figure 5.20](#), even without needing to create any extra places.

Zombie zone

1. Assuming $n \in \mathbb{N}$ is a natural number, what distinction is made by writing $n \langle \log n, 0 \rangle_{\delta_0^n}$ instead of just $n \log n$?
2. A pair of zombie processes X and Y are in a mutually recursive co-dependent relationship.

$$X \equiv \widetilde{\text{seq}}(\text{get } a, Y)$$

$$Y \equiv \widetilde{\text{seq}}(\text{put } b, X)$$



How could a zombie psychiatrist solve for each of them independently of the other using the **fix** combinator?

3. What is the general form of the solution for $X \equiv f(X, Y)$ and $Y \equiv g(X, Y)$? (hint: nested λ abstractions)
4. Give a formal specification for a transformation taking any well formed Petri net $x \in \mathbb{P}$ to a behavioral equivalent in $\widetilde{\mathbb{P}}$. Can it be made idempotent? How useful is it?
5. Would a definition of input completion (Equation 5.19) based on the configuration on the left of Figure 5.1 (instead of the right) interoperate badly with anything?
6. Could an alternative definition of mutual input completion (Equation 5.20) work by deleting vertices instead of creating them?
7. Even though the intuitive justification for the **fix** combinator invokes an assumption of functions expressible by process combinators, what happens in these cases? Do the results make any sense?
 - a) **fix** $\lambda x. x$
 - b) **fix** $\lambda x. C$
8. For what processes $C \in \widetilde{\mathbb{D}}$ does the behavioral equivalence $C \equiv \widetilde{\text{alt}}(C, C)$ hold?
9. What happens if the postset of an initially marked place in an operand to the $\widetilde{\text{alt}}$ combinator contains observable transitions?
10. What are the solutions to $X \equiv \widetilde{\text{alt}}(X, C)$ for a constant $C \in \widetilde{\mathbb{D}}$, and what is the Petri net model for the solution obtained by **fix** $\lambda x. \widetilde{\text{alt}}(x, C)$?
11. Is there any good reason for having the condition

$$\forall t \in T \cap \mathbb{T}. \mathcal{R}(A \cap (\{t\} \times \mathbb{V})) \oplus \mathcal{D}(A \cap (\mathbb{V} \times \{t\})) \neq \emptyset$$

on page 91 instead of just $T \cap \mathbb{T} \subseteq \mathcal{D}(A) \cup \mathcal{R}(A)$? (hint: item 6, page 194)

Each success only buys an admission ticket to a more difficult problem.

Henry Kissinger

CHAPTER **6**

REACHABILITY GRAPH WRANGLING

In this chapter, we develop reachability graphs in preparation for the transducers and finite automaton models of DI processes to follow in [Chapter 7](#). Together, these constructions allow a formal definition of a refinement relation as motivated in Part I, and also complete the program started in [Chapter 5](#) to define a robust core of process combinators. Whereas up to this point some process combinators have been limited to operating on processes modeled by open Petri nets (*i.e.*, members of $\tilde{\mathbb{D}}$) the ability to transform any open or closed Petri net model to an equivalent open canonical form effectively relieves this restriction. This capability is helpful among other reasons because it allows the `env` combinator ([Equation 5.21](#)) to be used liberally along with the others to write circuit specifications, even though its result is closed in general.



The initial construction of the reachability graph is fairly straightforward, with the remainder of this chapter given over to various possible graph transformations and optimizations. These are not sought for their own sake, but for the benefit of more efficient circuit synthesis and verification methods. Doing more work to simplify the reachability graph leaves less work to be done on simplifying the transducer, whose complexity directly impacts that of the synthesized circuit. Whether transducer optimizations alone could compensate is an open question. Our present purpose is only to promote flexibility in the implementation of a development tool chain.

6.1 Math usage

In principle, any of the models constructed in this chapter and the next can be defined by first order logic and set comprehension alone, but the nature of the material makes it worthwhile to

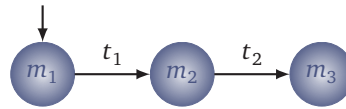


Figure 6.1: a reachability graph

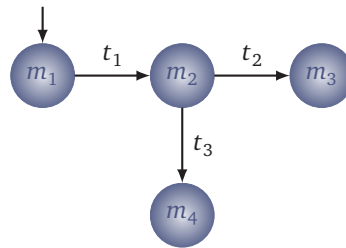


Figure 6.2: another reachability graph

extend the usual notation of mathematics temporarily with a limit construction and an assortment of strategically chosen higher order functions. Doing so also allows for a presentation supportive of software tool development without being overly prescriptive. Most of the relevant notation and conventions are introduced subsequently on the fly after a brief head start in this section.

6.1.1 Graphs

As a reminder, a reachability graph maps the space of markings a Petri net can attain in any number of steps by starting from a known initial marking and firing an enabled transition at each step. Two examples are shown in [Figure 6.1](#) and [Figure 6.2](#).

- The reachability graph in [Figure 6.1](#) describes a Petri net that can start with a marking m_1 where a transition t_1 is enabled. If t_1 fires, then the marking changes to m_2 with t_2 enabled. If t_2 fires when the marking is m_2 , the marking changes to m_3 , whereupon no transitions are enabled.
- The reachability graph in [Figure 6.2](#) describes a different Petri net, wherein two transitions t_2 and t_3 are both enabled in the marking m_2 . If t_2 fires, the marking changes to m_3 , but if t_3 fires, the marking changes to m_4 .

Although doing so may be intuitively helpful, drawing pictures of reachability graphs gets us only so far. To express algorithms for constructing and transforming reachability graphs, we need to be able to describe graphs more formally in general.

Describing a graph

The conventional method of specifying a graph entails a pair of sets (V, E) with V being the **vertices** in the graph, and $E \subseteq V \times V$ the **edges**. According to this convention, the graph in [Figure 6.1](#) is

described by the pair

$$(\{m_1, m_2, m_3\}, \{(m_1, m_2), (m_2, m_3)\})$$

which is to say $V = \{m_1, m_2, m_3\}$ is the set of vertices and $E = \{(m_1, m_2), (m_2, m_3)\}$ is the set of edges connecting them.

Time honored though it may be, this description of the graph neglects to indicate the transition that fires (e.g., t_1 , t_2 , or t_3) when the Petri net changes from one marking to the next, so it would be advantageous to take the liberty of extending it for our purposes. Instead of pairs (m_i, m_j) as above, edges could take the form of triples (m_i, t, m_j) , where m_i is the marking at the origin of the edge, t is the transition labeling the edge, and m_j is the marking at the terminus of the edge. According to this convention, the graph in Figure 6.2 would be represented as the following pair of sets (V, E) .

$$(\{m_1, m_2, m_3, m_4\}, \{(m_1, t_1, m_2), (m_2, t_2, m_3), (m_2, t_3, m_4)\})$$

However, having tampered to this extent with the usual style of describing a graph, we might as well go further by opting for a shorter one. The two edges (m_2, t_2, m_3) and (m_2, t_3, m_4) above have the same origin m_3 , so rather than repeating it, we could gather them into a single pair

$$(m_2, e) = (m_2, \{(t_2, m_3), (t_3, m_4)\})$$

and call $e = \{(t_2, m_3), (t_3, m_4)\}$ the **adjacency set** of m_2 . What we might term the adjacency set representation of a graph is then a set containing exactly one pair (m, e) for each vertex. For the graph in Figure 6.2, the adjacency set representation would be as follows,

$$\{(m_1, \{(t_1, m_2)\}), (m_2, \{(t_2, m_3), (t_3, m_4)\}), (m_3, \emptyset), (m_4, \emptyset)\}$$

where an empty adjacency set indicates an absence of outgoing edges from a vertex. Pairs of the form (m, \emptyset) are included nevertheless to avoid the need for a separate set V enumerating the vertices. The adjacency set representation for a graph is the preferred form used subsequently in this chapter.

Operating on graphs

Several ways of operating on graphs recur frequently enough to be worth adopting particular notations for them. These are summarized here for reference. The rest of this section may be worth a look even to readers conversant with graph theory because some of the notation is non-standard.



Reification As a set of pairs (m, e) with each m distinct, a graph in the adjacency set representation induces a function. Depending on who their high school teacher was, some readers might say it is a function, making the operator to be defined presently appear superfluous. While it is uncontroversial to identify a graph in adjacency set form as a relation, the convention in this book is to maintain a distinction between functions and relations regardless (following [19, 163, 271], but admittedly contrary to most introductory texts). When a graph g is to be used deliberately as a function, we denote the function $\Psi g : \mathcal{D}(g) \rightarrow \mathcal{R}(g)$ explicitly in terms of an operator Ψ defined as follows.

$$\Psi = \lambda g. \lambda m. \bigcup \mathcal{R}(g \cap (\{m\} \times \mathcal{R}(g))) \quad (6.1)$$

This operator is known by the quasi-standard term of **reification** hereafter. To keep score, we are up to one informal and two formal equivalent descriptions of a graph as shown in Figure 6.3.

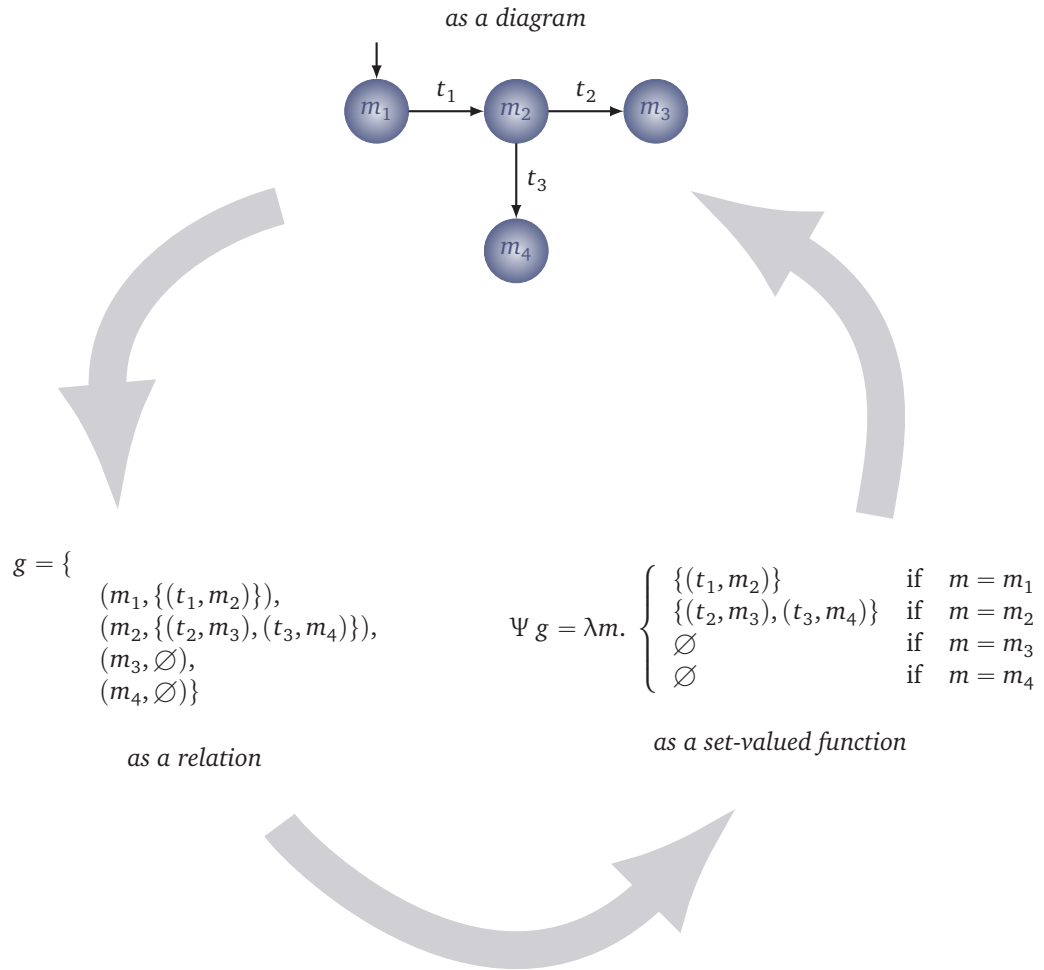


Figure 6.3: three equivalent descriptions of the reachability graph in Figure 6.2

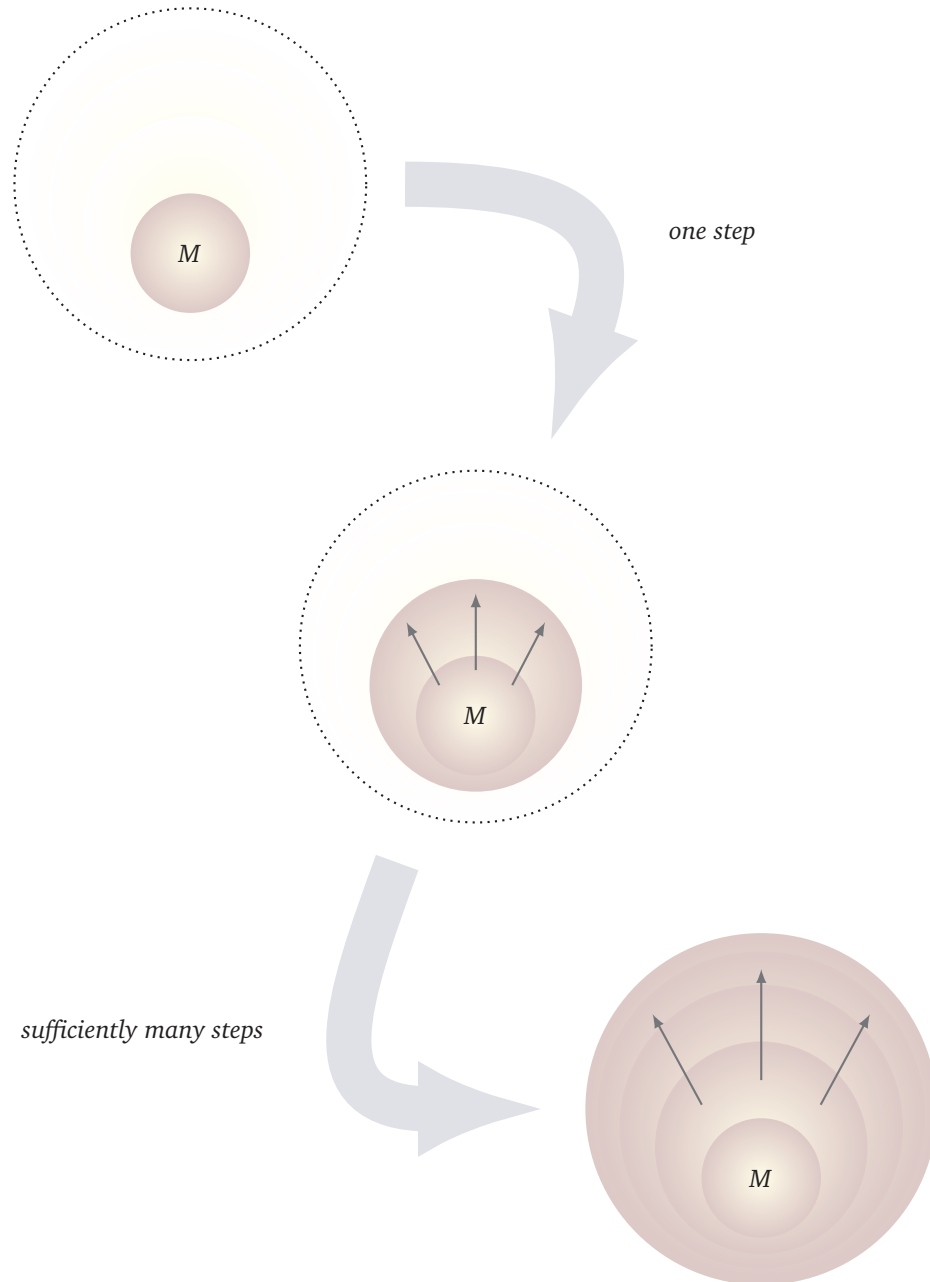


Figure 6.4: The space of known reachable markings is extended to its limit by starting with an initial marking M and engulfing the adjacent markings at each step (cf. Algorithm 4.1).

Percolation An initial vertex and some procedure for constructing adjacent vertices should normally suffice to determine a whole connected graph by iterating the procedure. The idea depicted intuitively in Figure 6.4 and indicated in somewhat greater detail by Algorithm 4.1 is not especially difficult to encapsulate as follows by yet another “polymorphic function” that turns out to be useful in various contexts.

First, a function f composed with itself i times for a natural number $i \in \mathbb{N}$ is expressed by f^i and defined according to this recurrence.

$$f^i(x) = \begin{cases} x & \text{if } i = 0 \\ f(f^{i-1}(x)) & \text{otherwise} \end{cases} \quad (6.2)$$

Next, a function f composed with itself infinitely many times is expressed as f^∞ and defined roughly as follows in terms of the notation above.

$$f^\infty = \lim_{n \rightarrow \infty} f^n = f \circ f \circ f \cdots \quad (6.3)$$

The expression $y = f^\infty x$ can be read constructively as the result of iterating f starting from x until a fixed point is reached (if one exists). Platonistically, it is the unique y from which $f^n x$ differs by less than any arbitrarily small $\epsilon > 0$ for all but finitely many $n \in \mathbb{N}$. For discrete set-valued functions f such as those that concern us in this chapter, the measure of the cardinality $|y \ominus f^n x| < \epsilon$ of their symmetric difference is more than adequate to make sense of this definition.

With this preparation and a known universe of vertices v , we can define a function

$$\rho : (v \rightarrow \mathcal{P}(v)) \rightarrow (\mathcal{P}(v) \rightarrow \mathcal{P}(v))$$

such that a function $f : v \rightarrow \mathcal{P}(v)$ mapping a given vertex $(m, e) \in v$ to a set of adjacent vertices $f(m, e) \in \mathcal{P}(v)$ determines a function $(\rho f) : \mathcal{P}(v) \rightarrow \mathcal{P}(v)$ that takes a non-empty subset $g \in \mathcal{P}(v)$ of a graph to the subgraph $(\rho f) g \in \mathcal{P}(v)$ reachable from members of g .

$$\rho = \lambda f. (\lambda g. g \cup \bigcup (\mu f) g)^\infty \quad (6.4)$$

There is no standard term for the ρ operator to the author’s knowledge. Maybe it should be called **percolation**, which is as good a name for it as any.

Pruning How many of these operators can be squeezed meaningfully into a single expression, and would defining one more provide enough ways of operating on graphs to get us started? Percolating a function that maps the reification Ψg of a graph g over the range $\mathcal{R}(e)$ of the adjacency set e of each member $(m, e) \in g$ might be expected to rebuild the whole graph when starting from the initial marking M . An operator Γ_M induced by any chosen marking M and defined by

$$\Gamma_M = \lambda g. (\rho \lambda(m, e). (\mu \lambda m'. (m', (\Psi g) m')) \mathcal{R}(e)) (g \cap \{(M, (\Psi g) M)\}) \quad (6.5)$$

effects this procedure. It should also work if M is not in g , in which case the result is \emptyset .

Clearly this operator satisfies $\Gamma_M g = \Gamma_M^2 g = \cdots = \Gamma_M^\infty g$, but not necessarily $g = \Gamma_M g$, which is what makes it useful. If a graph g has been transformed so as to delete any of the edges, then a vertex in g could lose all of its incident edges as a result, making it unreachable. A more parsimonious equivalent to g would be a subset omitting the unreachable vertices. We can read the expression $\Gamma_M g$ hereafter as the graph g pruned to the vertices reachable from M by way of any number of intermediate vertices.



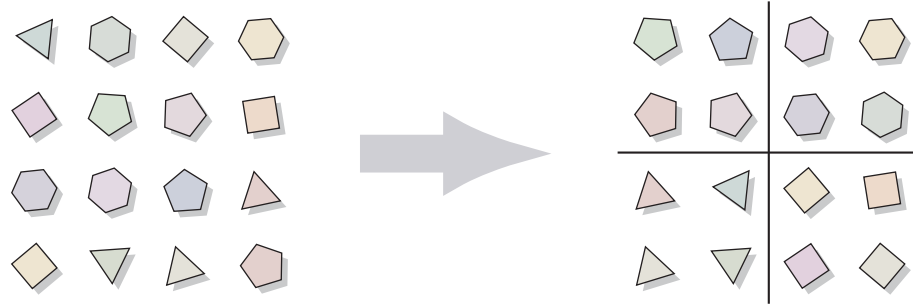


Figure 6.5: A set of shapes s partitioned by their number of sides is given by $(\pi f) s$ where f is the function that maps any shape to its number of sides.

6.1.2 Partitions

Following convention, a set $p \subset \mathcal{P}(s)$ is called a **partition** on s if $\bigcup p$ is equal to s and all sets $c \in p$ are pairwise disjoint. Each set $c \in p$ is called an **equivalence class**. Any function f defined on s induces a partition $(\pi f) s$ whereby any two members $q, r \in s$ belong in the same class c whenever the condition $f(q) = f(r)$ holds. The partitioning operator π consistent with this understanding can be defined as follows.

$$\pi = \lambda f. \lambda s. \{c \in \mathcal{P}(s) \mid ((\mu f) c) \cap (\mu f)(s - c) = \emptyset \wedge |(\mu f) c| = 1\} \quad (6.6)$$

Intuitively, f can be viewed as the function extracting the characteristic that members of the same class have in common with one another. For example, the partition on a set of shapes shown in Figure 6.5, for which any shapes in the same class always have the same number of sides, is determined by a function f that maps each shape to its number of sides. That is $f(x)$ is 3 if x is a triangle, 4 if x is a square, etc..

A related transformation on sets of pairs (x, y) is occasionally useful. For such a set s , the expression Πs denotes the set of all pairs (x, Y) where Y is the set of all right sides y of pairs $(x, y) \in s$ whose left side is x . For example, a set $s = \{(1, a), (1, b), (2, c), (2, d)\}$ would imply a value of $\Pi s = \{(1, \{a, b\}), (2, \{c, d\})\}$. The presence of one pair in the result Πs for each possible left side x in the given set s suggests a definition for Π that starts with a partition and then forms the union of a singleton set derived from each class.

$$\Pi = (\lambda p. \bigcup_{c \in p} \mathcal{D}(c) \times \{\mathcal{R}(c)\}) \circ \pi \lambda(x, y). x \quad (6.7)$$

6.1.3 Ordinals

The ordinal function notation defined in Section 5.1.4 is used extensively in this chapter on sets s outside of $\mathcal{P}(\mathbb{N})$ and $\mathcal{P}(\mathbb{V})$. We refrain hereafter from explicitly specifying the ordering on a set s inducing an ordinal function s° if it is covered by any of the following cases, which apply inductively.

- If s is a subset of \mathbb{N} , the usual ordering on natural numbers applies.
- If s is a subset of Petri net vertices, the ordering postulated by definition of \mathbb{V} applies.

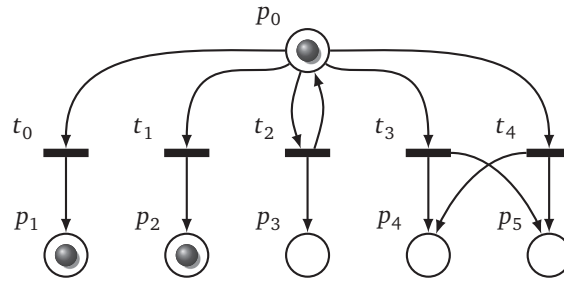


Figure 6.6: a Petri net with an initial marking

- A member (x, y) of a product of totally ordered sets precedes another member (x', y') if x precedes x' or if x is equal to x' and y precedes y' .
- Subsets of a totally ordered set are ordered such that
 - no set precedes itself
 - the empty set precedes any non-empty set
 - and a set u precedes a set v if either $\min(u)$ precedes $\min(v)$, or their minima are equal and $u - \{\min(u)\}$ precedes $v - \{\min(v)\}$.

6.2 Initial reachability graph

To keep the problem manageable, the reachability graph is constructed in several stages starting with the easy one in this section, but not all difficulties can be deferred. At a minimum, the formal definition of a Petri net $(P, T, A, M, F) \in \mathbb{P}$ developed in [Chapter 5](#) is a prerequisite. The initial and final markings M and F along with any other reachable markings m are subsets of the places $P \subset \mathbb{V}$, so it is natural to use a representation

$$g \in \mathcal{P}(\mathcal{P}(\mathbb{V}) \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}(\mathbb{V}))) \quad (6.8)$$

for the reachability graph in adjacency set form as planned in [Section 6.1.1](#). Here \mathbb{T} is the universe of observable transitions as always.

6.2.1 Overview

Although this representation is a good start, we can go no further without making some provision for unsafe markings. As the example in [Figure 6.6](#) shows, the transitions t_0 and t_1 are enabled because their preset $\{p_0\}$ is marked, but unsafe because their respective postsets $\{p_1\}$ and $\{p_2\}$ are also marked. The reachability graph should indicate that these transitions are enabled by exhibiting edges labeled by them, but where in the graph should these edges point? A temporary solution is to make their terminus a marking denoted

$$\perp_p = \{\min(\mathbb{V} - P)\} \quad (6.9)$$

consisting of the singleton set of the first vertex in \mathbb{V} that can not appear in any marking of a Petri net with places P . The vertex with the marking \perp_p can serve as the terminus of any edge associated with an unsafe transition. It necessarily has an empty postset, enables no transitions, and therefore can have no successors in the graph. This condition is appropriate because exploring the successors of unsafe markings yields no further information about the process being modeled. The solution is “temporary” because a way of removing this vertex without altering the trace semantics after the graph has been built is discussed in [Section 6.3](#).

Another technicality we can not afford to ignore pertains to the transition t_2 in [Figure 6.6](#), whose preset and postset both contain the place p_0 . The transition is enabled because its preset is marked, but is nevertheless safe even though a place in its postset is marked. When t_2 fires, a token is both debited and credited to p_0 atomically, which is not considered a safety violation. This example is a reminder that the safety condition for a transition t and a marking m is not that the postset $t \bullet$ is disjoint from m , but that the difference $t \bullet - \bullet t$ between the postset and the preset is.

One more point of detail is exemplified by the transitions t_3 and t_4 in [Figure 6.6](#). Both are enabled, at most one can fire, and the same successor marking ensues in either case. This example is a reminder that the successor marking does not uniquely determine the transition leading to it, so whatever method we use to enumerate successor markings when building the graph had best take note of their associated transitions as well.

6.2.2 Derivation

The reachability graph generating function ultimately takes the form of a percolation from the initial marking to the whole graph. Building the reachability graph of a process $X = (I, O, N) \in \mathbb{D}$ with a Petri Net model $N = (P, T, A, M, F) \in \mathbb{P}$ depends on being able to identify the enabled transitions with respect to an arbitrary marking $m \subseteq P$. To this end, the set T of transitions and the inverse



$$\bigcup_{(a,b) \in A \cap (P \times T)} \{(b, a)\}$$

of the adjacency relation A determine a set of pairs $(t, p) \in (J_0 A) T$ by a function

$$J_0 : \mathcal{P}(T \cup \mathbb{V}) \times \mathcal{P}((T \times \mathbb{V}) \cup (\mathbb{V} \times T)) \rightarrow \mathcal{P}((T \cup \mathbb{V}) \times \mathcal{P}(\mathbb{V})) \quad (6.10)$$

defined as

$$J_0 = \lambda(T, A). \bigcup_{t \in T - \mathcal{R}(A)} \{(t, \emptyset)\} \cup \Pi \bigcup_{(a,b) \in A \cap (\mathcal{D}(A) \times T)} \{(b, a)\}$$

wherein each left side t is a transition in the Petri net N and the corresponding right side $p = \bullet t$ is its set of preset places by [Equation 6.7](#). Hence a transition $t \in T$ is enabled with respect to a marking $m \subseteq P$ if and only if there is a pair $(t, p) \in (J_0 A) T$ for which p is a subset of m , or equivalently if the following conditions holds.

$$(t, p) \in J_0(T, A) \cap (T \times \mathcal{P}(m))$$

For a transition t having a preset p enabled with respect to a marking m , the set of postset places

$$t \bullet = s = (\Psi \Pi A) t$$

contributes to the successor marking $(m - p) \cup s$ unless t is unsafe due to m intersecting $s - p$ as noted previously. For unsafe transitions, we identify a fictitious marking \perp_p with the successor by [Equation 6.9](#), implying a result expressible in either case as

$$(\lambda i. \langle \perp_p, (m - p) \cup s \rangle_i) \delta_{\emptyset}^{m \cap (s - p)}.$$

Accordingly, the adjacency set e for a vertex (m, e) with a marking m in the reachability graph contains an edge

$$(t, (\lambda i. \langle \perp_p, (m - p) \cup s \rangle_i) \delta_{\emptyset}^{m \cap (s - p)}) \in e$$

for each transition t enabled with respect to m , or more explicitly

$$(t, (\lambda s. (\lambda i. \langle \perp_p, (m - p) \cup s \rangle_i) \delta_{\emptyset}^{m \cap (s - p)}) (\Psi \Pi A) t)$$

determining the whole adjacency set

$$\bigcup_{(t,p) \in J_0(T,A) \cap (T \times \mathcal{P}(m))} \{(t, (\lambda s. (\lambda i. \langle \perp_p, (m - p) \cup s \rangle_i) \delta_{\emptyset}^{m \cap (s - p)}) (\Psi \Pi A) t)\}$$

for that vertex, which we can abbreviate as $((J_1 J_0) (P, T, A)) m$ in terms of a function

$$J_1 = \lambda j. \lambda (P, T, A). \lambda m. \bigcup_{(t,p) \in j(T,A) \cap (T \times \mathcal{P}(m))} \{(t, (\lambda s. (\lambda i. \langle \perp_p, (m - p) \cup s \rangle_i) \delta_{\emptyset}^{m \cap (s - p)}) (\Psi \Pi A) t)\}$$

Then if we write $h(m)$ for the unit set $\{(m, e)\}$ of the vertex with marking m and its adjacency set e in terms of a function temporarily denoted

$$h = \lambda m. \{(m, ((J_1 J_0) (P, T, A)) m)\}$$

the whole reachability graph follows as

$$(\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} h n) (h M)$$

from the initial marking M and [Equation 6.4](#) in a definition of a reachability graph generating function

$$\mathbf{RG}_0 : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}(\mathbb{V})))$$

given by

$$\mathbf{RG}_0 = \lambda(I, O, (P, T, A, M, F)). (\lambda h. (\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} h n) (h M)) \lambda m. \{(m, ((J_1 J_0) (P, T, A)) m)\}.$$

6.3 Divergence propagation

Evaluating $\mathbf{RG}_0(X)$ for a process $X = \mathbf{get} a$ results in the reachability graph shown in [Figure 6.7](#), where the vertices are visualized as translucent orbs revealing within each of them the marking it contains. The vertex labeled M pertains to the initial marking and the one labeled F pertains to the final marking.

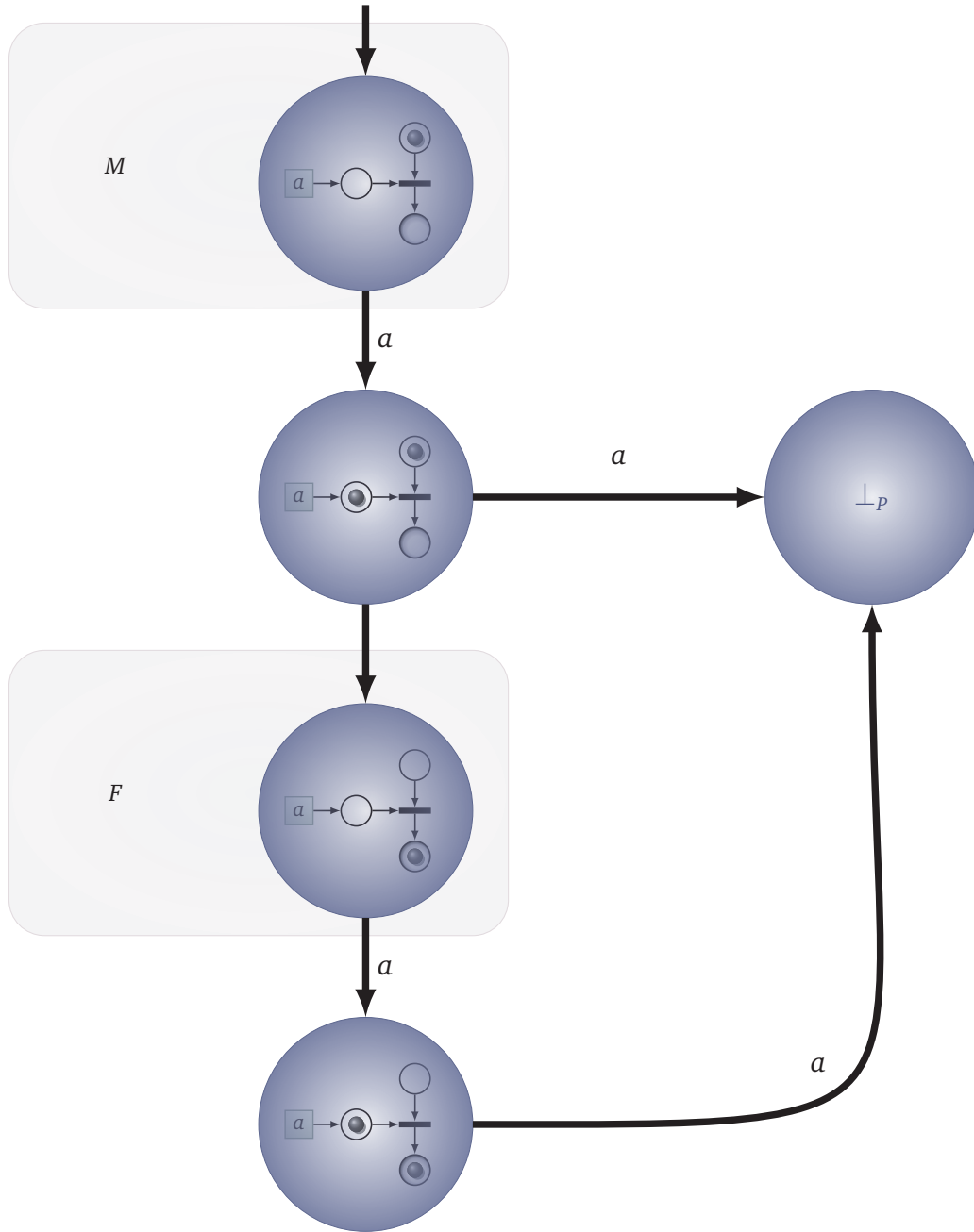
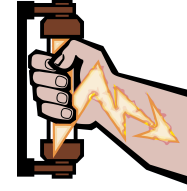


Figure 6.7: the initial reachability graph $RG_0(X)$ with $X = (I, O, (P, T, A, M, F)) = \text{get } a$

This example raises a couple of puzzling questions. Why should the “final” marking have a successor? More troublingly and not unrelated, does the process accept an input of a twice, or just once? On one hand, after the first input of a , the time will come for the internal transition to fire, whereupon a second input of a will be safe. On the other hand, the internal transition is unobservable, so it will never be safe to assume that time has come. Clearly something that might be unsafe is as bad as something that is unsafe, so the correct interpretation is that it accepts the input only once, but what rewrite rule in general would remove only the spurious edges like the second a in Figure 6.7 from a reachability graph without ever mistakenly altering the process semantics?

Puzzling indeed, let us hold that thought and stick to something easier for now, which is to get rid of the vertex labeled \perp_p as mentioned in Section 6.2.1.



6.3.1 Divergent vertices

If we are going to get rid of \perp_p , then it is only fair to get rid of other markings that are no better. Any vertex (m, e) with a marking m and an adjacency set e containing an edge (t, \perp_p) is as unsafe as \perp_p itself if t is an anonymous transition or if t is an output, because it means the Petri net having attained the marking m can devolve spontaneously to \perp_p beyond the control of the environment.

Finding these stealthily unsafe markings is easy by walking backwards from \perp_p through the reachability graph $g = \mathbf{RG}_0(I, O, N)$ along any edges not labeled by inputs in I . A graph like g with the directions of the edges reversed and the input labeled edges deleted

$$\prod_{(m,e) \in g} \prod_{(t,n) \in e - (I \times \mathcal{R}(e))} \{(n, (t, m))\}$$

pruned to the vertices reachable from \perp_p contains only unsafe markings in its domain

$$\mathcal{D}(\Gamma_{\perp_p} \prod_{(m,e) \in g} \prod_{(t,n) \in e - (I \times \mathcal{R}(e))} \{(n, (t, m))\})$$

which we can denote for the moment as $Y_0(I, P) \ g \in \mathcal{P}(\mathcal{P}(\mathbb{V}))$ in terms of a function

$$Y_0 : \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{V}) \rightarrow (\mathcal{P}(\mathcal{P}(\mathbb{V})) \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}(\mathbb{V}))) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{V}))$$

defined by

$$Y_0 = \lambda(I, P). \lambda g. \mathcal{D}(\Gamma_{\perp_p} \prod_{(m,e) \in g} \prod_{(t,n) \in e - (I \times \mathcal{R}(e))} \{(n, (t, m))\}).$$

6.3.2 Disabled inputs

It would appear justifiable to rewrite every member of $Y_0(I, P) \ g$ to \perp_p wherever it occurs in any marking or its adjacency set throughout the graph g , and even to empty the adjacency sets of markings thus rewritten by leaving their vertices as (\perp_p, \emptyset) because their successor markings are of no practical consequence, but that transformation in itself would not rid the graph entirely of \perp_p as planned. To justify excising the sole surviving instance of (\perp_p, \emptyset) and all edges reaching it, we may reason as follows.

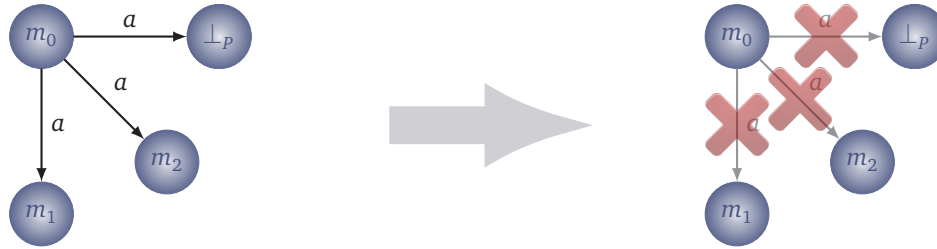


Figure 6.8: Deleting an edge labeled a from m_0 to the divergent vertex labeled \perp_P without changing the semantics requires deleting the edges labeled a from m_0 to safe markings m_1 and m_2 .

- In an open Petri net-modeled DI process, every input transition is always enabled because its preset is empty, whereas in a closed one, which can only be the form $\mathbf{env}(X, E)$, an input is disabled when the specification restricts the environment E from emitting it.
- In the reachability graph, a disabled input with respect to a marking m is indicated by the lack of an outgoing edge from the vertex (m, e) labeled by that input.
- The effect of a disabled input in a closed Petri net model is to relieve the process of any obligations in the event of its reception. Hence its behavior is undefined.
- Therefore a disabled input is equivalent to an enabled input that causes divergence.
- In a reachability graph where all vertices containing an unsafe marking are rewritten to (\perp_P, \emptyset) , no vertex has an outgoing edge connected to (\perp_P, \emptyset) unless that edge is labeled by an input, because if it were labeled by an output or an anonymous transition, the vertex would have been transformed to (\perp_P, \emptyset) itself.
- If an edge from a vertex (m, e) to (\perp_P, \emptyset) is labeled by an input a , then deleting all edges labeled a from (m, e) disables a with respect to m , and because a has been deemed to cause divergence, disabling it makes no semantic difference.
- All incident edges to (\perp_P, \emptyset) can be deleted without changing the semantics, thereby making it unreachable and appropriate for deletion.

The only slippery part of this argument concerns the case of an input a labeling more than one edge from the same vertex, of which only one points to \perp_P . To disable it, we have to delete not only the one pointing to \perp_P , but the others as well, which may point to perfectly safe vertices. This situation is depicted in [Figure 6.8](#).

It should be clear nevertheless that deleting all similarly labeled edges from a vertex is the right thing to do when one of them leads to divergence. The reachability graph in [Figure 6.8](#) describes a Petri net that may choose non-deterministically among three alternatives when it receives an input of a after having attained the marking m_0 . If one of those alternatives is unsafe, and the environment has no control over which alternative is chosen, then the input a must be regarded as unsafe at this point. If only the edge leading to divergence were deleted and the others were allowed to remain, then the semantics would be altered to express that the input a is safe.

With that settled, there is no impediment to eliminating all explicit references to unsafe markings at once by replacing the graph g with one like

$$g - ((y \ g) \times \mathcal{R}(g))$$

where $y \ g = Y_0(I, P) \ g$ contains all unsafe markings in g , along with removing the “dangling references” to the erstwhile vertices in

$$\bigcup_{(m,e) \in g - ((y \ g) \times \mathcal{R}(g))} \{(m, e - (\mathcal{D}(e) \times (y \ g)))\}$$

and topping it off with a prune in

$$\Gamma_M \bigcup_{(m,e) \in g - ((y \ g) \times \mathcal{R}(g))} \{(m, e - (\mathcal{D}(e) \times (y \ g)))\}$$

to eliminate any vertices that may have been safe themselves but reachable only through unsafe vertices, and therefore are not reachable any more. Here M is the initial marking of the Petri net model $N = (P, T, A, M, F)$ in the process $X = (I, O, N)$ whose reachability graph is $g = \mathbf{RG}_0(X)$ as usual. This improved graph is still subject to further improvements, but worth denoting for future reference as $((Y_1 \ M) \ Y_0(I, P)) \ g$ in terms of a function

$$Y_1 = \lambda M. \lambda y. \lambda g. \Gamma_M \bigcup_{(m,e) \in g - ((y \ g) \times \mathcal{R}(g))} \{(m, e - (\mathcal{D}(e) \times (y \ g)))\}.$$

6.3.3 Numbered vertices

From this point onwards, much of the process semantics can be inferred from the way the reachability graph is connected without reference to the Petri net from which it is derived, so there is little reason to maintain a record of its markings. A marking $m \in \mathcal{P}(\mathbb{V})$ for every vertex $(m, e) \in g$ seems like excess baggage when a graph of the form

$$g \in \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

with numeric values in place of markings should serve just as well (cf. Equation 6.8). A full account of the process semantics (*i.e.*, one that would enable the recovery of an equivalent Petri net model to the original from the reachability graph) would require identifying one vertex in the reachability graph with the initial marking and one with the final marking if any, but this need could be met by a numbering scheme that always maps the initial marking M to 1 and final marking F to 0 (or the opposite, but this way turns out to be more convenient for reasons that are presently impossible to motivate). Our last task in this section therefore is to tidy up the reachability graph representation accordingly.

A numbering function

Any numbering function that takes a marking m to δ_m^M whenever m is a member of $\{M, F\}$ would be adequate, with the numbers assigned to other markings unconstrained, so we can expect it to be some function of the form

$$\lambda m. (\lambda i. \langle \delta_m^M, 2 + f \ m \rangle_i) \delta_{\{M, F\}}^{\{M, F\} - \{m\}}$$

where the index

$$i = \delta_{\{M,F\}}^{\{M,F\}-\{m\}}$$

is unity whenever m is neither M nor F , and f is some numbering function applicable to all other markings m . The part about f is not difficult either, because the set of all other markings in the graph g is expressible as

$$\mathcal{D}(g - (\{M, F\} \times \mathcal{R}(g)))$$

and a function that maps each member of this set to a unique natural number is expressible in terms of the ordinal function notation defined in [Section 5.1.4](#).

$$f = (\mathcal{D}(g - (\{M, F\} \times \mathcal{R}(g))))^\circ$$

Hence our whole numbering function for a graph g with initial and final markings M and F can be denoted $Y_2(M, F) g$ in terms of a function Y_2 defined as follows.

$$Y_2 = \lambda(M, F). \lambda g. \lambda m. (\lambda i. \langle \delta_m^M, 2 + (\mathcal{D}(g - (\{M, F\} \times \mathcal{R}(g))))^\circ m \rangle_i \delta_{\{M,F\}}^{\{M,F\}-\{m\}})$$

Anonymous edge unification

Applying this function to every marking in the graph g leaves no trace of the markings in their original concrete representation as sets of Petri net places, so the unobservable Petri net transitions in \mathbb{V} labeling the anonymous edges can have no further significance other than as place holders. Further transformations in [Section 6.4](#) and [Section 6.6](#) are simplified by erasing distinctions among them, for which it suffices to map every remaining anonymous edge label $t \in \mathbb{V}$ to the same minimum member of \mathbb{V} in the course of renumbering the vertices as part of a combined transformation $Y_3 Y_2(M, F)$ with Y_3 given by

$$Y_3 = \lambda y. \lambda g. \bigcup_{(m,e) \in g} \{((y g) m, \bigcup_{(t,n) \in e} \{(\lambda i. \langle t, \min \mathbb{V} \rangle_i \delta_{\emptyset}^{\{t\}-\mathbb{V}}, (y g) n\})\})\}.$$

An aptly named reachability graph generating function to improve on \mathbf{RG}_0 would be

$$\mathbf{RG}_1 : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

showing Y_0 through Y_3 together in the context of an explicit formal parameter $X \in \mathbb{D}$.

$$\mathbf{RG}_1(X) = (\lambda(I, O, (P, T, A, M, F)). (Y_3 Y_2(M, F)) ((Y_1 M) Y_0(I, P)) \mathbf{RG}_0 X) X$$

6.4 Anonymous edge reduction

Although the work of [Section 6.3](#) reduces the reachability graph in [Figure 6.7](#) from five vertices to four as shown in [Figure 6.9](#), it does not address the issues raised at the outset regarding trace semantics and final markings. With no resolution at hand, perhaps simplifying the graph further may yield some insight.



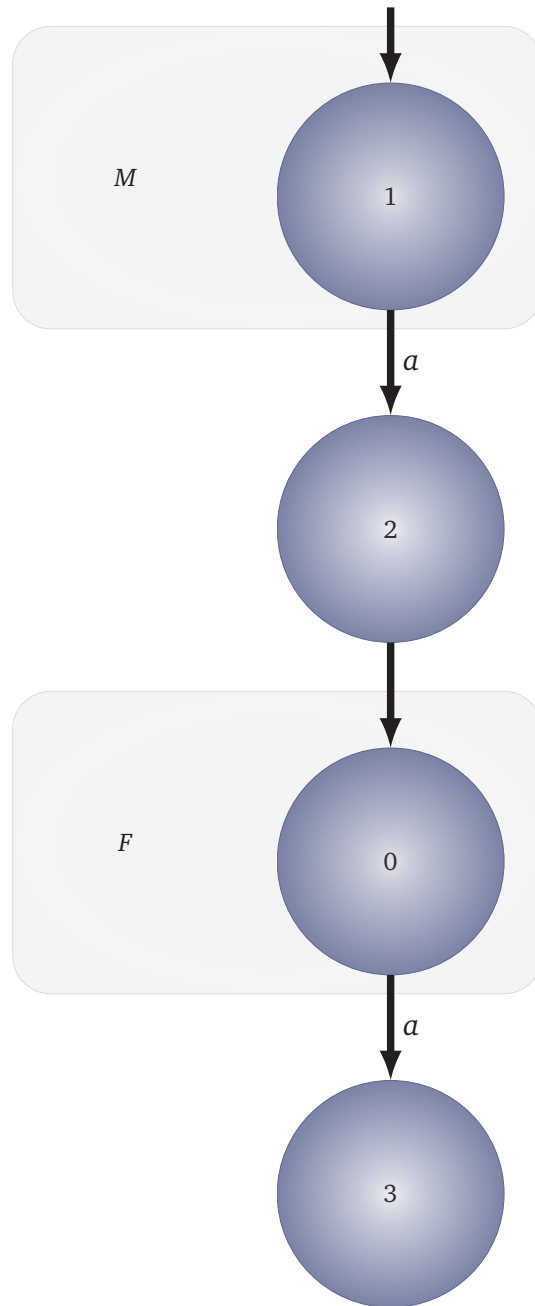


Figure 6.9: revised reachability graph $\mathbf{RG}_1(X)$ with $X = (I, O, (P, T, A, M, F)) = \mathbf{get} \ a$

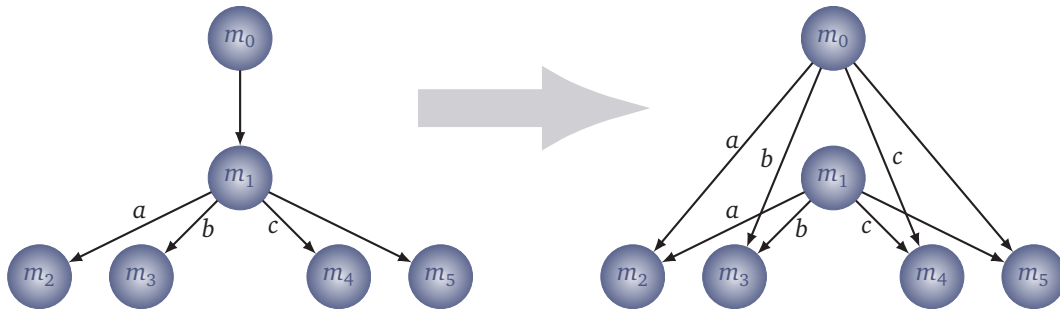


Figure 6.10: An anonymous edge from m_0 to m_1 is replaced by edges from m_0 directly to the termini of all edges originating from m_1 and labeled similarly.

6.4.1 Overview

An obvious way to simplify a reachability graph is shown in Figure 6.10. If the Petri net can slip silently from a marking m_0 to m_1 by firing an unobservable transition, then for practical purposes it is as if anything that can happen when the marking is m_1 can happen when the marking is m_0 , so why not bypass m_1 and connect m_0 directly to its successors m_2 through m_5 ? If m_1 becomes unreachable as a result, so much the better because the graph then can be simplified by pruning it. By the time this transformation is applied throughout the graph, maybe all the edges labeled by unobservable transitions will be eliminated along with quite a bit of dead wood.

There is unfortunately something wrong with this idyllic picture. If the observable transitions a , b , and c are outputs, then there is no problem, but if any of them is an input, this transformation alters the semantics. In the graph at the left, none of them is enabled with respect to m_0 , so based on the reasoning in Section 6.3.2, any of them that is an input causes divergence if it is received when the marking is m_0 . In the graph subsequent to the transformation shown at the right, *voilà*, they are all safe. This change is significant. The edge from m_0 to m_1 in the original graph implies that they eventually become safe, but because there is no way to discern when they do, they must always be considered unsafe.

Theoretically this sword should cut both ways. If a vertex numbered m with an enabled input a connects by an anonymous edge to a successor m' where a is disabled, that means a is temporarily safe while the marking is m but eventually becomes unsafe when the marking changes to m' , which the environment can not observe or control. Perhaps a should be considered unsafe with respect to the current marking m as well. While it is straightforward to delete edges found to correspond to unsafe inputs by this criterion from a reachability graph, it is unnecessary because they can not occur in a reachability graph whose Petri net is expressible by the process combinators developed in Chapter 5. This effect is by design. It is incompatible with delay insensitivity for a process to disable its own inputs without intervention from the environment [288].

Alternatively, inputs that are temporarily disabled before spontaneously becoming enabled certainly can occur in reachability graphs derived from valid DI process specifications, this characteristic being evident in Figure 6.9. Graphs like this one can be simplified without altering their semantics by a slightly restricted version of the transformation illustrated in Figure 6.10. The restricted transformation refrains from creating any outgoing edges from vertices numbered m that are labeled by inputs disabled with respect to m , but otherwise proceeds as shown. Transforming the graph in

Figure 6.9 by this rule would delete the edge from vertex 2 to vertex 0, but would not create an edge labeled a from 2 to 3 because a is disabled with respect to vertex 2.

Not only would this transformation make the graph possible to prune by excising vertices 0 and 3, but it would also allow the elimination of the edge labeled a from vertex 0 to vertex 3. This outcome is desirable because it would make the trace semantics readily apparent from the graph. The input a is acceptable only once, and exactly one edge labeled a would remain. This effect is not a coincidence, but exactly the right way for spurious edge labels like the latter a to be removed.

However, having stumbled onto this solution to one of the problems first observed in Section 6.3, we may have created a worse problem. Previously the final marking F may have had an unwanted successor, but at least there was a final marking. Following the current plan would leave the graph without one because the vertex numbered 0 would be pruned. Losing track of the final marking obliterates potentially useful semantic information. To keep it from being lost, we have to follow this transformation by another rewrite rule: if there is no final marking in the graph after pruning, then any surviving vertices that were originally connected to vertex 0 through a path of anonymous edges are identified with it by renumbering them to 0. If there are more than one of them, the renumbering effectively fuses them, but let us ignore this situation for the moment.

According to this new plan, the graph in Figure 6.9 would be transformed to two vertices, with vertex 0 deleted and vertex 2 renumbered to 0 to represent the final marking, which is an improvement. We are left with a graph of one vertex for the initial marking, one for the final marking, and an input-labeled edge between them, fortunately just as it should be.

Maybe not so fortunately, this last manipulation unhelpfully rewards a *cavalier* attitude. It is not generally valid to apply this transformation when the terminus of the anonymous edge to be deleted has no reachable successors. A vertex (m, \emptyset) with no successors expresses deadlock. A vertex (m', e) with several outgoing edges and one anonymous edge $(t, m) \in e$ to a vertex (m, \emptyset) with no successors expresses a process with an option to deadlock at its own discretion. Taking away that option changes the semantics. We might not like it, but suppressing it artificially amounts to killing the messenger. The need to specify the exact conditions whereby the transformation preserves semantics calls for a more formal discussion.

6.4.2 Derivation

A fair amount of technical detail warrants breaking a derivation of anonymous edge reduction into several parts retracing the ideas above more systematically. Traversing the graph exhaustively through only the anonymous edges starting at each vertex essentially creates all necessary additional edges at once, but eliminating the edges thus made redundant is complicated by considerations of final markings and deadlock semantics, which have to be discussed at some length as a prerequisite. When these matters are resolved, attention to an anomalous corner case prolongs the discussion again, but at last we arrive at another revision of the reachability graph with most if not all anonymous edges removed.

Traversing anonymous edges

To start with the obvious, let a reachability graph g restricted to its anonymous edges

$$(\mu \lambda(m, e). (m, e \cap (\mathbb{V} \times \mathcal{R}(e)))) g$$

induce a function taking a vertex number n to the set of numbers of vertices reachable from the vertex numbered n through any number of anonymous edge traversals

$$\lambda n. \mathcal{D}(\Gamma_n(\mu \lambda(m, e). (m, e \cap (\mathbb{V} \times \mathcal{R}(e)))) g)$$

denoted $Z_0 g$ for $Z_0 : \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow (\mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}))$ defined by

$$Z_0 = \lambda g. \lambda n. \mathcal{D}(\Gamma_n(\mu \lambda(m, e). (m, e \cap (\mathbb{V} \times \mathcal{R}(e)))) g).$$

Creating new edges

Any node $(m, e) \in g$ with an adjacency set e is connected by a sequence of anonymous edges to every vertex numbered $n \in (Z_0 g) m$, whose collective adjacency sets are

$$d = \bigcup_{n \in (Z_0 g) m} (\Psi g) n \in \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}).$$

It would be tempting to enlarge the adjacency set e to $e \cup d$ so as to bypass the intermediate vertices along the anonymous edge paths as in [Figure 6.10](#), but following from the discussion of disabled inputs above, we should at least restrict it to

$$e \cup d - ((I - \mathcal{D}(e)) \times \mathcal{R}(d))$$

where I is the input alphabet of the process whose reachability graph is g , so as not to enable any currently disabled inputs in $I - \mathcal{D}(e)$. Although it should never be necessary for a valid DI process specification, we can also explicitly disable transiently enabled inputs as an extra sanity check following the earlier discussion by rewriting the adjacency set e to

$$e \cup d - ((I - (\mathcal{D}(e) \ominus \mathcal{D}(d))) \times \mathcal{R}(e \cup d))$$

where the symmetric difference $\mathcal{D}(e) \ominus \mathcal{D}(d)$ contains any edge labels in one of e or d but not the other. In this way, we decouple the reachability graph from complicated assumptions about the Petri net model. Let the rewritten graph up to this point be denoted $((Z_1 I) Z_0) g$ according to the following definition.

$$Z_1 = \lambda I. \lambda z. \lambda g. (\mu \lambda(m, e). (\lambda d. (m, e \cup d - ((I - (\mathcal{D}(e) \ominus \mathcal{D}(d))) \times \mathcal{R}(e \cup d)))) \bigcup_{n \in (z g) m} (\Psi g) n) g$$

Identifying final marking equivalents

The next step should be to eliminate the anonymous edges that are now redundant in the graph. If a vertex $(m, e) \in g$ is connected by an anonymous edge $(t, m') \in e$ to another vertex $(m', e') \in g$ where the adjacency set e' of the terminus is a subset of the adjacency set e of the origin, then it would seem reasonable to eliminate the edge (t, m') from e , but as noted previously we can not do so indiscriminately. For one thing, we have to keep the vertex representing the final marking from becoming unreachable unless we can designate others to take its place.

Candidates to replace the final marking vertex would be those connected to it by an anonymous edge, which are members of $Z_2 g$ with Z_2 defined explicitly for the sake of concreteness as

$$Z_2 = \lambda g. \{m \in \mathcal{D}(g) \mid (\Psi g) m \cap (\mathbb{V} \times \{0\}) \neq \emptyset\}.$$

In the unlikely case of multiple vertices indicated by $Z_2 g$ with conflicting adjacency sets, it is best to refrain from using any of them as replacements and to leave the original vertex 0 in place along with any anonymous edges leading to it. Hence we look for a cardinality $|i| = 1$ in the set

$$i = (\mu \Psi g) Z_2 g$$

of adjacency sets of members of $Z_2 g$ as a prerequisite (relying implicitly on anonymous edge unification as proposed in Section 6.3.3). A further condition necessary for preserving semantics when members of $Z_2 g$ are rewritten to 0 should be

$$(\bigcup i) - (\mathbb{V} \times \{0\}) \subseteq (\Psi g) 0$$

meaning that none of them indicates any behavior contrary to that of the original 0 numbered vertex (at least in the context of a graph already transformed by $(Z_1 I) Z_0$). A restriction $(Z_3 Z_2) g$ to a set that either meets these conditions or devolves to $\{0\}$ follows from Z_3 given by

$$Z_3 = \lambda z. \lambda g. (\lambda i. (\lambda j. (\lambda k. \langle \{0\}, z g \rangle_k) \delta_1^{|i|} \delta_j^{\emptyset})) (\bigcup i) - (\mathbb{V} \times \{0\}) - (\Psi g) 0) (\mu \Psi g) z g$$

and justifies fusing its members into a single new vertex numbered 0 while allowing the alternative to become unreachable.

Preserving deadlock semantics

We are still not ready to eliminate any anonymous edges from the graph until arranging to retain those terminating on vertices with no successors, which express deadlock as noted previously. For an adjacency set e in a vertex $(m, e) \in g$, we could eliminate every anonymous edge in e terminating on a vertex whose adjacency set is a subset of e as originally proposed, *except* the termini with empty adjacency sets, by rewriting e to

$$e - (\mathbb{V} \times \mathcal{D}(g \cap (\mathbb{N} \times (\mathcal{P}(e) - \{\emptyset\}))))$$

and if we can remember at the same time to avoid edge deletions that might make the vertex numbered by 0 or its replacement candidates $(Z_3 Z_2) g$ unreachable, then the given adjacency set e should be rewritten to

$$e - (\mathbb{V} \times \mathcal{D}(g \cap ((\mathbb{N} - (Z_3 Z_2) g) \times (\mathcal{P}(e) - \{\emptyset\}))))$$

which prevents eliminating any edge terminating at a vertex indicated by $(Z_3 Z_2) g$, whether 0 or its replacements. This transformation also depends on anonymous edge unification as mentioned above.

Eliminating redundant edges

Compared to the rewrite rule above for the adjacency set e , the one for the vertex number m is easy. If m is a member of $(Z_3 Z_2) g$, then it refers either to a vertex connected to the 0 vertex by an anonymous edge or to 0 itself, and can be rewritten to 0. A compact description of the rewritten vertex number

$$m \delta_{(Z_3 Z_2) g}^{\{(Z_3 Z_2) g\} - \{m\}}$$

(or possibly not) in the whole rewritten vertex

$$(m\delta_{(Z_3 Z_2)g}^{((Z_3 Z_2)g)-\{m\}}, e - (\mathbb{V} \times \mathcal{D}(g \cap ((\mathbb{N} - (Z_3 Z_2)g) \times (\mathcal{P}(e) - \{\emptyset\}))))))$$

leaves no further obstacle to eliminating redundant edges from a graph g by its transformation to $(Z_4 Z_3 Z_2) ((Z_1 I) Z_0) g$ in terms the following definition.

$$Z_4 = \lambda z. \lambda g. (\mu \lambda (m, e). (m\delta_{zg}^{(zg)-\{m\}}, e - (\mathbb{V} \times \mathcal{D}(g \cap ((\mathbb{N} - zg) \times (\mathcal{P}(e) - \{\emptyset\})))))) g$$

However, we are left with a minor anomaly when 0 is not a member of $z g = (Z_3 Z_2) ((Z_1 I) Z_0) g$ but the vertex numbered 0 in g has an empty adjacency set, which is not implausible. Excluding a number m from $z g$ allows but does not require anonymous edges terminating at m to be deleted, whereas having an empty adjacency set prohibits edges to it from being deleted. On the other hand, membership of m in $z g$ requires it to be rewritten to 0. These conditions imply an edge from a formerly m -numbered vertex pointing to the 0 vertex (or ambiguously to itself), and the original vertex $(0, \emptyset) \in g$ remaining distinct from that one. To correct this anomaly, we have to apply one further transformation that explicitly fuses all similarly numbered vertices by unifying their adjacency sets, forcibly removes any anonymous edge from a vertex to itself, and then takes the opportunity to prune all unreachable vertices, the last being worthwhile in any case.

$$Z_5 = \lambda g. \Gamma_1 (\mu \lambda m. (m, ((\Psi g) m) - (\mathbb{V} \times \{m\}))) \mathcal{D}(g)$$

We might be stuck with a few lingering anonymous edges in rare cases even after this transformation to the graph, but this last revision completes a new improved version $g = \mathbf{RG}_2 X$ of the reachability graph expressible in terms of a function

$$\mathbf{RG}_2 : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

given by

$$\mathbf{RG}_2(X) = (\lambda(I, O, N). Z_5 (Z_4 Z_3 Z_2) ((Z_1 I) Z_0) \mathbf{RG}_1 X) X.$$

6.5 Redundant path elimination

The next improvement to the reachability graph is motivated by the example shown in [Figure 6.11](#), which is based on a practical process specification discussed in [Section 4.1.1](#). According to the graph in [Figure 6.11](#), an initial input of R_1 followed immediately by an input of R_2 could lead either to a vertex where the system can only output G_1 , or to a vertex where the system must choose non-deterministically between the outputs G_1 and G_2 . Furthermore, if the system outputs G_1 , then it reaches the same vertex regardless of the path it takes to get there, which the user can neither influence nor observe.

This reachability graph is more complicated than necessary. If the environment transmits R_1 and then R_2 without waiting for an output inbetween, then whatever happens internally, the net effect is that an output of either G_1 or G_2 must be expected. If one path encompasses both alternatives, then the other path is redundant and can be eliminated without consequence. A similar argument applies to R_2 followed by R_1 and to both inputs concurrently. As far as the environment is concerned, the reachability graph might as well be the one shown in [Figure 6.12](#), which has two vertices and four edges fewer than that of [Figure 6.11](#).

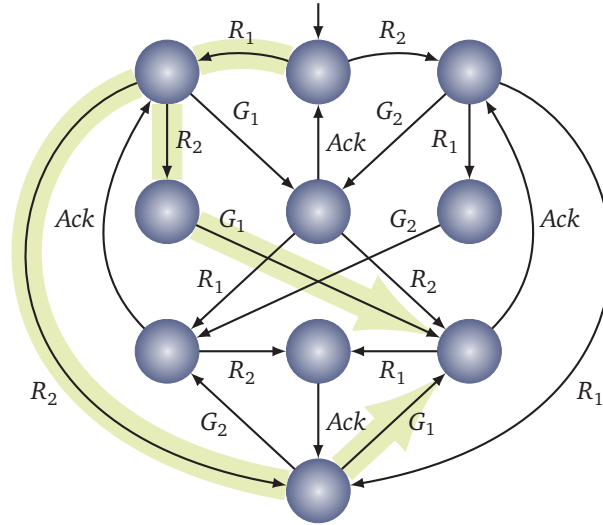


Figure 6.11: two paths, highlighted, with the same labels leading to the same vertex (cf. Figure 4.4)

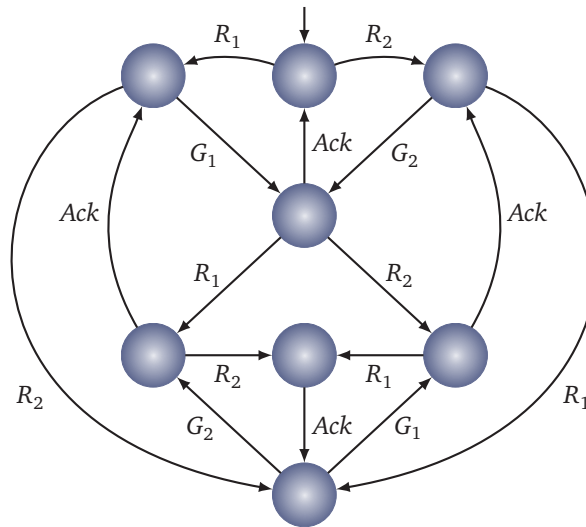


Figure 6.12: the reachability graph in Figure 6.11 reduced by redundant path elimination

A general formulation of this optimization is the subject of this section. As usual, we require it to preserve the trace semantics and the presence of a vertex numbered 0 associated with the final marking if any. A guaranteed optimal result would probably require a costly depth-first search of the graph, but the approach taken in this section procures an approximate solution efficiently by iterating a local rewrite rule. In some cases the rule creates new vertices in the hope of a net improvement by making others unreachable and thus subject to pruning, but each iteration must be conditional on a strict decrease in the number of edges or vertices in the graph to ensure termination. Further informal discussion of the transformation follows in [Section 6.5.1](#), and a derivation in detail follows in [Section 6.5.2](#).

6.5.1 Overview

To generalize from this example, a redundant path always starts where a vertex has multiple outgoing edges with identical labels, such as a member $(m, e) \in g$ of a graph g with edges

$$e = \{(t_0, m_0), (t_1, m_1)\}$$

satisfying $t_0 = t_1$. Maybe in this case either edge (t_0, m_0) or (t_1, m_1) is the beginning of a redundant path and can be eliminated from e , but undoubtedly some other conditions apply.

The redundancy of either path depends on what follows from it, which would depend on the outgoing edges from the vertices numbered m_0 and m_1 . A conservative assumption is that if the adjacency sets of m_0 and m_1 are identical, meaning $(\Psi g) m_0 = (\Psi g) m_1$ holds, then whatever can happen to m_0 could also happen to m_1 , so one of the edges would certainly be redundant. However, this criterion is too restrictive to be much help, and is covered in any case by partition fusion as described in [Section 6.6](#).

A less conservative assumption would be that if the adjacency set associated with m_0 is a subset of that of m_1 , meaning $(\Psi g) m_0 \subset (\Psi g) m_1$ holds, then any behavior associated with m_0 is included in that of m_1 , so the edge (t_0, m_0) is redundant and can be eliminated from e . Though superficially plausible, this proposition is not quite correct. If m_1 has an outgoing edge labeled with an input i_1 , but m_0 does not, then i_1 is enabled when the system attains m_1 , but not when it attains m_0 . Because being disabled is the same as being unsafe ([Section 6.3.2](#)), the input i_1 is not safe with respect to m_0 . If the system starts at m and undergoes a transition t equal to either t_0 or t_1 (which are equal to each other by hypothesis), then the safety of i_1 following t depends on whether the system reaches m_0 or m_1 . One can only assume that t followed by i_1 is not generally safe under these circumstances. Eliminating the path from m through m_0 would change the trace semantics by making it safe.

If $(\Psi g) m_0 \subset (\Psi g) m_1$ does not imply that the path through m_0 is redundant, does it imply instead that the path through m_1 is redundant? One might argue that it does because any inputs disabled in m_0 must be presumed unsafe, so it makes no difference if they are enabled in m_1 . However, there could be an outgoing edge from m_1 labeled by an *output* o_2 , but not from m_0 . Whether the output o_2 can occur following the transition t from m depends on whether the system reaches m_0 or m_1 . One must assume that it could happen. Eliminating the path through m_1 would change the semantics by asserting that it could not, so neither is this path redundant.

Although neither path by itself provides for the elimination of the other in this situation, there could be yet a third path making both of them redundant, which brings us to [Figure 6.13](#). This path would also have to originate at m and be labeled by t , but would pass through another vertex whose enabled inputs are the intersection of those of m_0 and m_1 , and whose enabled outputs are

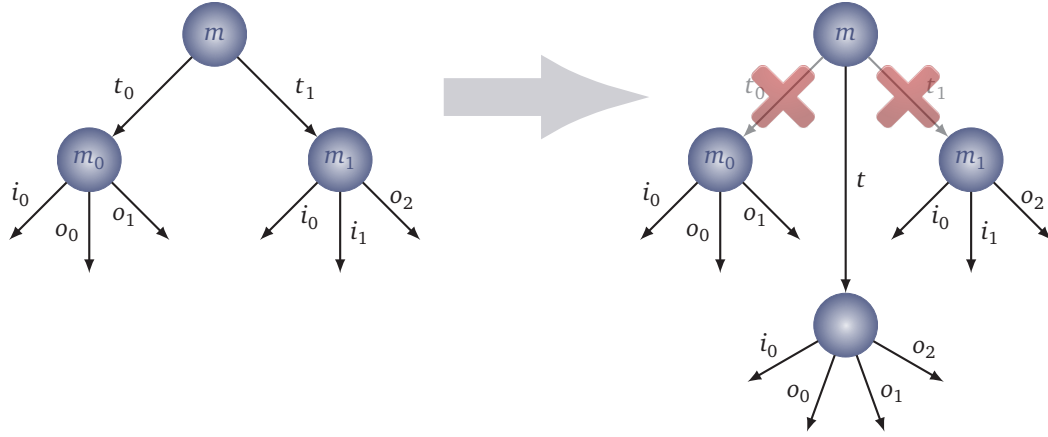


Figure 6.13: Neither extant path from m makes the other redundant, but a newly created vertex enables a path making both of them redundant.

the union of those from m_0 and m_1 . In this way, any inputs disabled from either of m_0 and m_1 are also disabled from this vertex, but any outputs enabled from either of them are enabled from it. This vertex need not be in the graph already; it might be a net gain to create it if doing so allows both m_0 and m_1 to be eliminated. This insight is enough to get started on the formal specification in the rest of this section.

6.5.2 Derivation

The relevance to this transformation of an adjacency set $e = (\Psi g) m$ in a vertex $(m, e) \in g$ having members (t, m_0) and (t, m_1) with different termini m_0 and m_1 but the same edge label t suggests that a partition $\Pi e = \{(t, \{m_0, m_1 \dots\}), \dots\}$ might be useful. With that, we could seek to rewrite the adjacency set e to a replacement with just one edge in it for each member $(t, s) \in \Pi e$, where the new edge has the original label t but a new terminus derived from the set s of original termini m_i . If the set $s = \{m_0\}$ contains only one member m_0 , then the edge remains unchanged as (t, m_0) by the transformation.

Last things first

With many details left to address, we can dispense fortunately with two of the more worrisome in one swoop at the outset, one being to ensure the continued reachability of the node representing the final marking, and the other being to preserve semantics pertaining to deadlock. By partitioning the set s in a pair $(t, s) \in \Pi e$ further according to the conditions

$$(\pi \lambda n. (\delta_{\emptyset}^{((\Psi g)^n)^{-I}}, \delta_0^n)) s$$

where I is the input alphabet of the process whose reachability graph g is being transformed, we ensure that any vertex numbered $n = 0$ is in a class by itself, and any vertices lacking outputs in their adjacency sets (hence exhibiting deadlocking or quiescent behavior) are separated from any

with outputs. The first of these conditions requires any edge of the form $(t, 0) \in e$ to induce a pair $(t, \{0\}) \in s$ rewritable to the same edge $(t, 0)$ as before, so that edges pointing to the 0 vertex never disappear. The other condition is motivated by the idea that the new vertex to be derived from s , if any, must have an output-labeled edge in its adjacency set matching each output-labeled edge in any member of s . A union of adjacency sets of which some contain output-labeled edges but others do not results in one that necessarily contains output-labeled edges, and therefore no longer expresses the option not to output, but keeping dissimilar adjacency sets separate prevents this misunderstanding (albeit at the possible cost of separately created vertices). To capture these ideas more formally, let $((W_0 I) g) m$ denote the set of pairs (t, s) associated with the vertex numbered m in the graph g with input alphabet I according to a function

$$W_0 : \mathcal{P}(\mathbb{T}) \rightarrow (\mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow (\mathbb{N} \rightarrow \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}(\mathbb{N}))))$$

defined as follows.

$$W_0 = \lambda I. \lambda g. \lambda m. \bigcup_{(t,s) \in \Pi(\Psi g) m} \{t\} \times (\pi \lambda n. (\delta_{\emptyset}^{((\Psi g)^n)^{-I}}, \delta_0^n)) s$$

Adjacency sets

On the subject of adjacency sets, the one associated with the set $s \in \mathcal{D}(g)$ of vertex numbers obtained by $(W_0 I) g$ above consists of two subsets, one being the intersection of the input-labeled edges in all adjacency sets $e \in (\mu \Psi g) s$, and the other being the union of all output-labeled or anonymous edges therein. The latter is easy to express formally as one might expect by

$$\bigcup_{e \in (\mu \Psi g) s} e - (I \times \mathcal{R}(e))$$

but the former is not quite the analogous cumulative intersection. If an input labels at least one edge in every adjacency set e in $(\mu \Psi g) s$, then that input is enabled with respect to the vertex to be created for s , so all edges labeled by it are allowed in the result even if no particular edge so labeled is common to every $e \in (\mu \Psi g) s$ due to varying termini. To capture this requirement precisely, we need to be clear about allowing the set of enabled inputs

$$i = \bigcap_{d \in (\mu \Psi g) s} \mathcal{D}(d) \cap I$$

to include any input common to the domains of all adjacency sets in $(\mu \Psi g) s$ regardless of the termini of the edges they label, so that any edge labeled by any of them

$$\bigcup_{e \in (\mu \Psi g) s} e \cap (i \times \mathcal{R}(e))$$

is allowed in the result. An expression $((W_1 I) g) s$ combining both subsets is then straightforward in terms of a function

$$W_1 : \mathcal{P}(\mathbb{T}) \rightarrow (\mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow (\mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})))$$

given by

$$W_1 = \lambda I. \lambda g. \lambda s. ((\lambda i. \bigcup_{e \in (\mu \Psi g) s} e \cap (i \times \mathcal{R}(e))) \bigcap_{d \in (\mu \Psi g) s} \mathcal{D}(d) \cap I) \cup \bigcup_{e \in (\mu \Psi g) s} e - (I \times \mathcal{R}(e)).$$

Vertex numbers

Having obtained an adjacency set $a = ((W_1 I) g) s$ as above, we lack only a vertex number $n \in \mathbb{N}$ for the new vertex (n, a) determined by s . The number n can be chosen arbitrarily as any non-member of $\mathcal{D}(g)$, but a value of

$$n = (\max \mathcal{D}(g)) + \mathcal{P}(\mathcal{D}(g))^\circ s$$

suffices to guarantee a unique assignment for any $s \in \mathcal{P}(\mathcal{D}(g))$. However, it would be wasteful to create a new vertex if there is already a member of g having exactly $a = ((W_1 I) g) s$ as its adjacency set, and even invalid to do so if $s = \{0\}$ refers to the final marking, whose number 0 must be kept invariant. To express the appropriate vertex number in either case, the relation

$$\{(a, (\max \mathcal{D}(g)) + \mathcal{P}(\mathcal{D}(g))^\circ s)\} \cup (\mu \lambda(m, e). (e, m)) g$$

consisting of the inverse of the reachability graph g and one additional related pair (a, n) determines a function

$$\Psi \Pi (\{(a, (\max \mathcal{D}(g)) + \mathcal{P}(\mathcal{D}(g))^\circ s)\} \cup (\mu \lambda(m, e). (e, m)) g)$$

taking any adjacency set in $\{a\} \cup \mathcal{R}(g)$ to a set of corresponding vertex numbers, from which the minimum given by

$$\min (\Psi \Pi (\{(a, (\max \mathcal{D}(g)) + \mathcal{P}(\mathcal{D}(g))^\circ s)\} \cup (\mu \lambda(m, e). (e, m)) g)) a$$

is always the best choice to associate with the given adjacency set a . Denote this result $(W_2 g) (a, s)$ in terms of a function $W_2 : \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow ((\mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}) \times \mathcal{P}(\mathbb{N})) \rightarrow \mathbb{N})$ defined by

$$W_2 = \lambda g. \lambda(a, s). \min (\Psi \Pi (\{(a, (\max \mathcal{D}(g)) + \mathcal{P}(\mathcal{D}(g))^\circ s)\} \cup (\mu \lambda(m, e). (e, m)) g)) a.$$

Rewrite rule

With this preparation, we can formulate the rewrite rule up to this point roughly in terms of the functions W_0 , W_1 , and W_2 . For every vertex number $m \in \mathcal{D}(g)$ and every pair (t, s) of edge labels t and sets s of termini in $((W_0 I) g) m$, the adjacency set $a = ((W_1 I) g) s$ and the new or used vertex number $n = (W_2 g) (a, s)$ determine a tuple $((m, (t, n)), (n, a))$ (yes, with two copies of n). The set of all such tuples

$$u = \bigcup_{m \in \mathcal{D}(g)} \bigcup_{(t, s) \in ((W_0 I) g) m} (\lambda a. (\lambda n. \{((m, (t, n)), (n, a))\})) (W_2 g) (a, s) ((W_1 I) g) s$$

determines a pair $(d, r) = (\Pi \mathcal{D}(u), \mathcal{R}(u))$ of sets of partly rewritten vertices where all adjacency sets in d are up to date but some in r may be stale. Before addressing this last issue, let us summarize this result as $(d, r) = W_3 \langle (W_0 I, W_1 I), W_2 \rangle g$ with W_3 defined as follows.

$$W_3 = \lambda(w, v). \lambda g. (\lambda u. (\Pi \mathcal{D}(u), \mathcal{R}(u))) \bigcup_{m \in \mathcal{D}(g)} \bigcup_{(t, s) \in (w_0 g) m} (\lambda a. (\lambda n. \{((m, (t, n)), (n, a))\})) (v g) (a, s) (w_1 g) s$$

Staleness

The issue of stale adjacency sets in the result (d, r) obtained by W_3 stems from vertices $(n, a) \in r$ that are also members of the previous iteration of the graph g because a matching adjacency set

$a \in \mathcal{R}(g)$ is found by W_2 . If the adjacency set corresponding to the vertex number n is rewritten to something other than a in d , then there are two vertices both numbered n in with different adjacency sets in d and r . Although the adjacency sets are semantically equivalent, clearly the one in d should take precedence or else the effort to rewrite it is wasted. A revised reachability graph

$$d \cup (r - (\mathcal{D}(d) \times \mathcal{R}(r)))$$

includes all newly created vertices from r along with their adjacency sets but ignores any members of r whose current or updated versions appear in d . Some members of d should also be ignored henceforth, namely those that have been made unreachable due to edges deleted from the adjacency sets. This further step is expressible as $W_4(d, r)$ by a transformation

$$W_4 : \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \times \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

defined as suggested above with an additional pruning operation.

$$W_4 = \lambda(d, r). \Gamma_1 (d \cup (r - (\mathcal{D}(d) \times \mathcal{R}(r))))$$

Iteration

It is not unlikely that transforming the graph g by $w = W_4 \circ W_3(\langle W_0 I, W_1 I \rangle, W_2)$ would lead to a result enabling further improvements by the same transformation, and that the limit $w^\infty g$ would be the most improved of all, but some restraint is necessary because this limit need not exist. The issue at stake is illustrated in [Figure 6.14](#). Visible at the upper left, a vertex with two outgoing edges labeled a connects to two vertices each having an outgoing edge labeled b . In the first step, a new vertex is created on that basis, which necessarily has two outgoing edges labeled b , as shown in the center image. The newly created vertex allows another opportunity to apply the rewrite rule. This time, a new vertex is created requiring two outgoing edges labeled a , as shown at the lower right. The graph is now back where it started except for the two new vertices. This process obviously can continue indefinitely.

Fortunately, this issue is not difficult to remedy once recognized. To guarantee termination, each application of the rewrite rule to a graph g must be made conditional on the strict decrease of a metric $\|g\|$ chosen as any measure of the size of the graph that can not become arbitrarily small. A natural choice would be the sum of the number of vertices and the number of edges in the graph.

$$\|g\| = |g| + \sum_{(m,e) \in g} |e|$$

Other reasonable alternatives are the number of vertices only or the number of edges only. The specific choice is largely an implementation decision best guided by its effects in typical production settings.

A modified rewrite rule that alters the graph only when doing so implies a net improvement in the size metric is expressible as $W_5 (W_4 \circ W_3(\langle W_0 I, W_1 I \rangle, W_2))$ based on a function W_5 given by

$$W_5 = \lambda w. \lambda g. (\lambda i. \langle w g, g \rangle_i) (\lambda m. \delta_m^{\|g\|}) \min \{ \|g\|, \|w g\| \}$$

This rule always reaches a fixed point when iterated exhaustively.

Not a moment too soon, the next major edition of the reachability graph for a process $X \in \mathbb{D}$ becomes $\mathbf{RG}_3(X)$ in terms the previous edition $\mathbf{RG}_2(X)$ and a function

$$\mathbf{RG}_3 : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

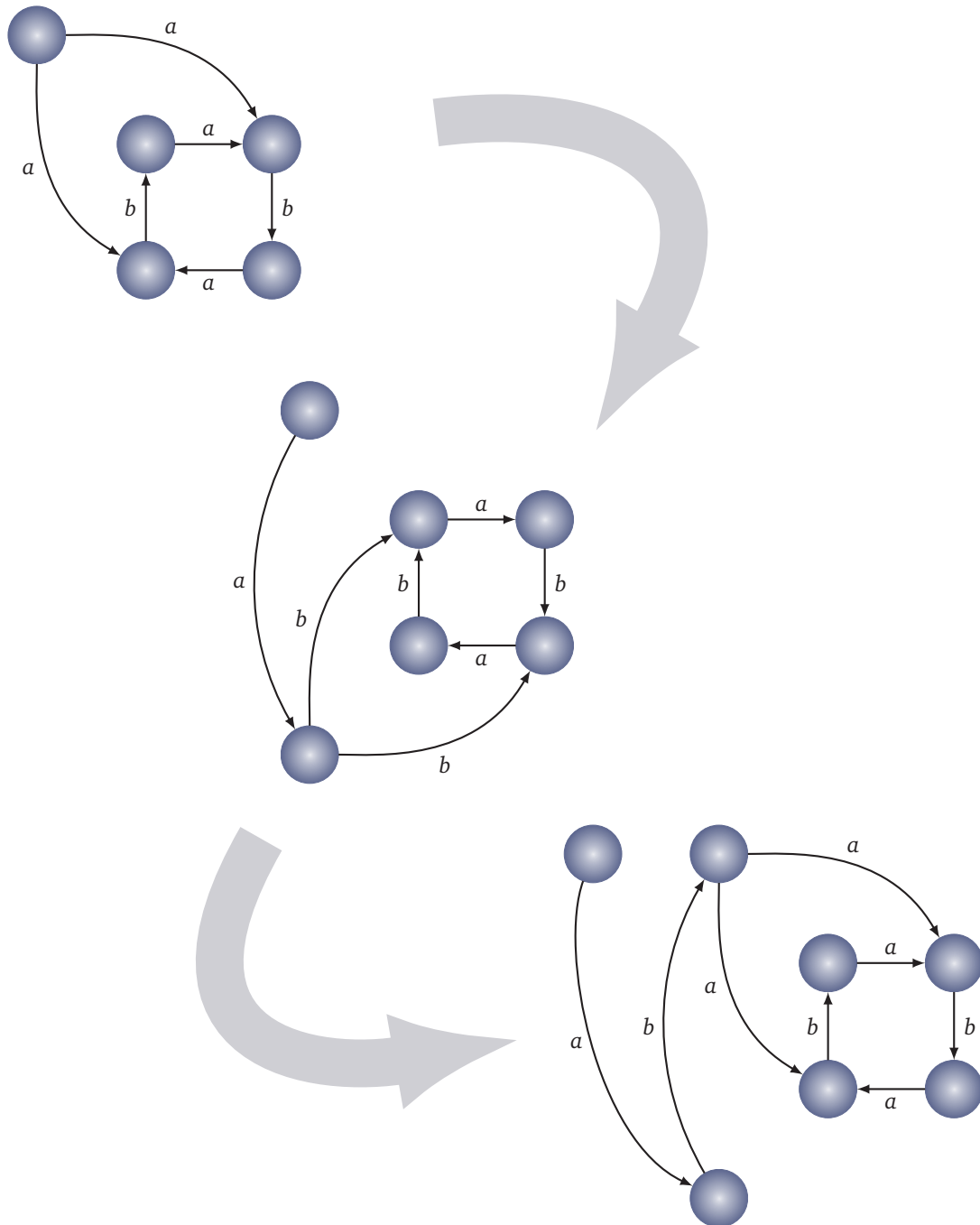


Figure 6.14: For some graphs, a redundant edge removal transformation can create new vertices *ad infinitum*, which is undesirable.

defined as follows.

$$\mathbf{RG}_3(X) = (\lambda(I, O, N). (W_5 (W_4 \circ W_3 (\langle W_0 I, W_1 I \rangle, W_2)))^\infty \mathbf{RG}_2 X) X$$

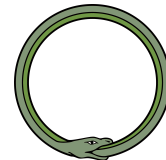
6.6 Partition fusion

The final transformation of the reachability graph considered in this chapter is motivated by [Figure 6.15](#). Why should we tolerate the tangled mess at the top of the figure when the nice clean alternative at the bottom obviously does the same job of expressing the cyclic repetition of four signals a , b , c , and d ? There is no good reason, especially when a well known algorithm for state machine minimization is readily adaptable to the problem of converting one to the other [115].

6.6.1 Overview

The core idea of the transformation is that if some of the vertices in a graph are behaviorally equivalent to one another, then they can be merged into a single vertex. In [Figure 6.15](#), the vertices of the upper graph form natural equivalence classes based on similarity of outgoing edge labels. However, similar edge labels alone are not generally sufficient for equivalence, because the edges could lead to different destination vertices having different observable behavior. In this example, the edges from any two “equivalent” vertices do indeed lead to different destination vertices. Nevertheless, the destinations always fall within the same equivalence class, which is crucial. For two vertices to be behaviorally equivalent to each other, they must not only have the same set of labels on their outgoing edges, but have behaviorally equivalent destination vertices for each label as well.

An insightful reader might perceive an impasse at this point. To decide whether two vertices are behaviorally equivalent to each other requires a determination as to whether the vertices at the other ends of their outgoing edges are behaviorally equivalent. Determining *their* behavioral equivalence in turn requires looking still further ahead. A correct partition would seem to be justifiable only in retrospect, having been found only by a costly exhaustive search.



An iterative algorithm

Nevertheless, there is an efficient algorithm for obtaining the desired partition that avoids infinite regress. The algorithm starts with a course approximation to the partition determined only by edge labels, and refines it iteratively until a termination condition is satisfied.

- The termination condition requires that if the edges labeled t from a vertex in a class c are connected to vertices in classes $d_0, d_1 \dots d_n$, then every other vertex in c also has at least one outgoing edge labeled t connected to some vertex each class d_i , and no edges labeled t to vertices in any classes other than those.
- The operation performed when the condition is not met is to find a class c in which an edge labeled t can lead from one member to a class d but not from another member, and then to subdivide p so that the members connecting to d are assigned to one subdivision and rest are assigned to the other.

This operation must be iterated until the condition is met because subdividing any class may require classes containing predecessors of its members also to be subdivided, which may require subdivisions

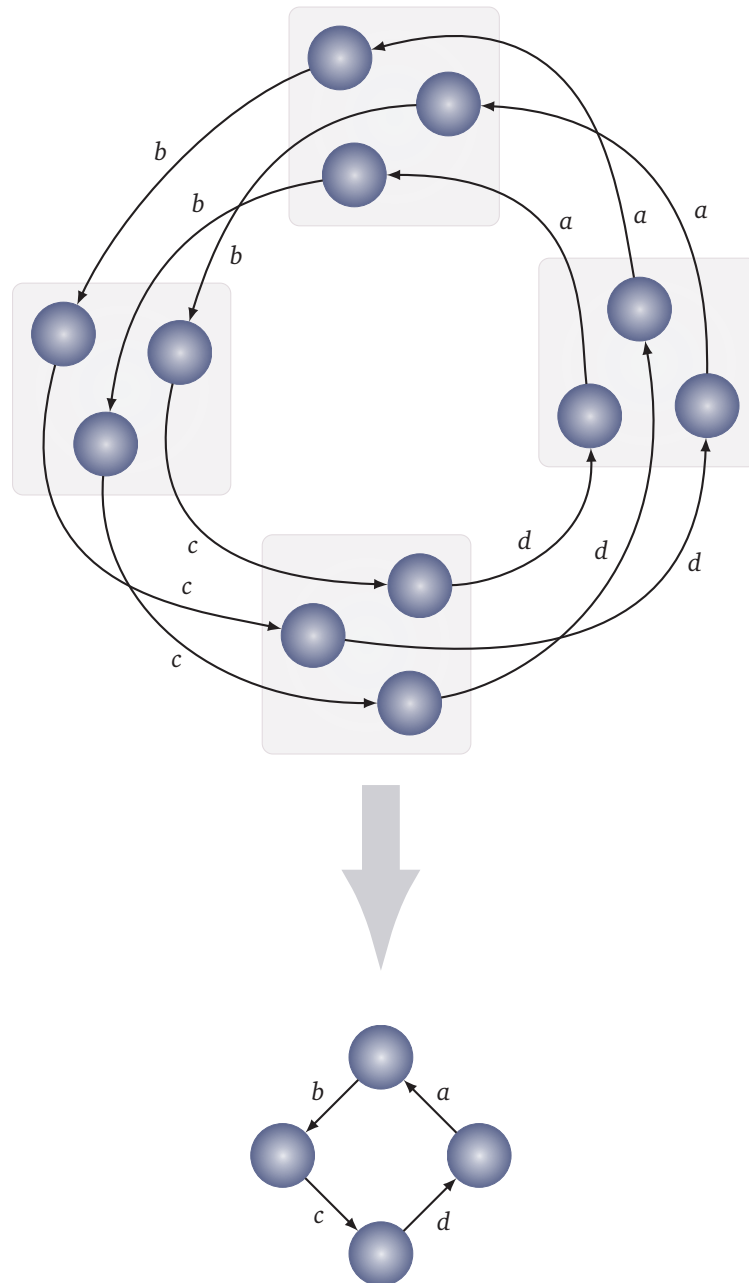


Figure 6.15: Partition fusion reduces each equivalence class (above, shaded) to a single vertex (below).

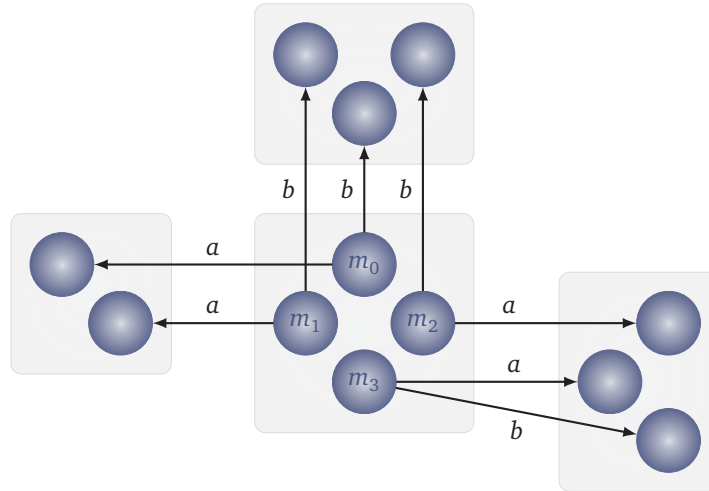


Figure 6.16: In the initial partition, m_0 through m_3 are classed together because they each have an outgoing edge labeled a and one labeled b .

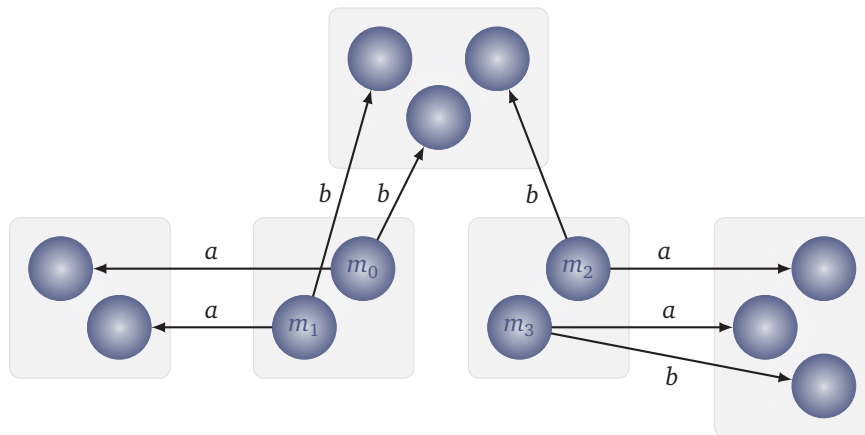


Figure 6.17: Because edges labeled a from m_0 and m_1 lead to a different class than those from m_2 and m_3 in Figure 6.16, m_0 and m_1 have to be put into a different class than m_2 and m_3 .

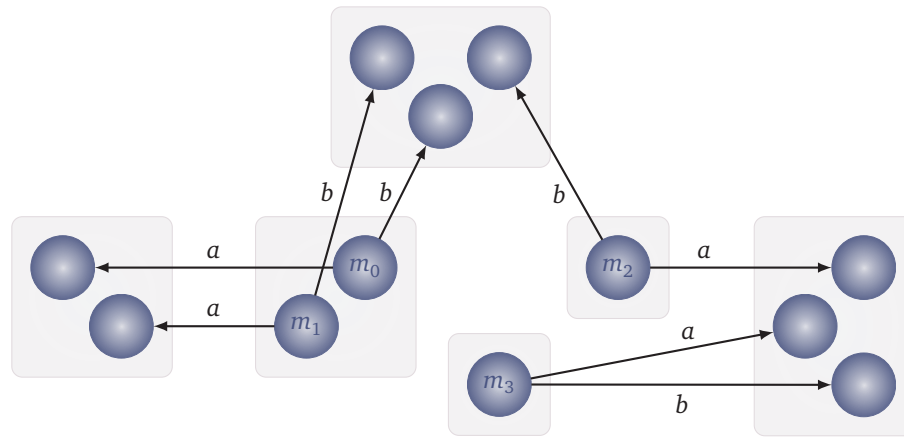


Figure 6.18: The class containing m_2 and m_3 in Figure 6.17 has to be subdivided further because their edges labeled b lead to different classes.

of other classes, and so on. However, termination can be guaranteed because the condition is trivially satisfied if every class is reduced to a single vertex. To complete the transformation after the termination condition is satisfied, every equivalence class in the graph is transformed to a single vertex.

A partial example

Three snapshots of the graph transformation are shown in Figure 6.16 through Figure 6.18. These may be helpful for visualizing this algorithm in action.

- In Figure 6.16, a graph is partitioned initially with respect only to the edge labels, resulting in several classes, including one containing the vertices m_0 through m_3 . These vertices are in the same class because each of them has an outgoing edge labeled a and an outgoing edge labeled b . Outgoing edges from vertices in other classes are not shown.
- In Figure 6.17, the former class containing m_0 through m_3 has been subdivided into one for m_0 and m_1 , and another for m_2 and m_3 . This operation is indicated because edges labeled a from m_0 and m_1 lead to vertices in the class on the left, whereas edges labeled a from m_2 and m_3 lead to vertices in the class on the right.
- The former class containing m_2 and m_3 is further subdivided in Figure 6.18, because the outgoing edges labeled b from these vertices go to different classes.

Not represented in the figures are possible knock-on effects from these subdivisions. Any other classes with vertices connected by similar labels to different members of the class originally containing m_0 through m_3 may need to be subdivided due to this class being subdivided.

Final markings

The preservation of a vertex numbered 0 to be identified with the final marking is more straightforward for this transformation than for previous ones. Mainly it involves an assignment of the number 0 to the vertex identified with the class containing vertex 0 when the classes are converted to vertices at the end. There is one potential pitfall in the case of a graph with distinct initial and final vertices being transformed to one in which they end up in the same equivalence class. Although it is not clear that such a graph could emerge from a valid DI process specification, a provision to avoid this outcome is nevertheless part of the formal specification discussed next.

6.6.2 Derivation

To start somewhat from the bottom up, much of this transformation, known hereafter as **partition fusion**, entails a representation of a graph as a set g of equivalence classes c whereby the usual graph representation as a set of vertices is recoverable as $\bigcup g \in \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$. It is helpful in this context to be able to write $(P_0 g) m \in \mathcal{P}(\mathbb{N})$ for the set of all vertex numbers belonging to vertices in the same class c as that of a vertex numbered by a given $m \in \mathcal{D}(\bigcup g)$. This function is easily expressible as

$$P_0 = \lambda g. \Psi \bigcup_{c \in g} \mathcal{D}(c) \times \{\mathcal{D}(c)\}.$$

The part about subdividing the partition described above then permits a straightforward expression as $(P_1 P_0) g$ for a graph in this representation with P_1 defined as follows.

$$P_1 = \lambda p. \lambda g. \bigcup_{c \in g} (\pi \lambda(m, e). (\mu \lambda(t, n). (t, (p g) n)) e) c$$

That is, each vertex (m, e) in a set c is equivalent only to the others in that set for which every outgoing edge (t, n) labeled by a transition t in its adjacency set e points to a member of the same set $(P_0 g) n$ of vertex numbers as the one containing n . If two members of any set $c \in g$ are not equivalent by this criterion, they separate in $(P_1 P_0) g$, and if their separation causes others not to belong together, the rest all separate eventually in $(P_1 P_0)^\infty g$.

This step would usually suffice to determine the reduced graph were it not for the requirement to preserve individually identifiable initial and final vertices, numbered 1 and 0 respectively by convention. If both of these vertices land in the same class due to being otherwise equivalent, we must separate them by hand in $P_2 (P_1 P_0)^\infty g$ according to

$$P_2 = \lambda g. \bigcup_{c \in g} (\pi \lambda(m, e). \delta_0^m \delta_{\mathcal{D}(c)}^{\mathcal{D}(c) \cup \{1\}}) c$$

which puts the final vertex numbered $m = 0$ in a class by itself only if it has shared a class hitherto with the initial one. This action may affect the equivalence of other vertices connected to either of them, so another round of subdivisions $(P_1 P_0)^\infty P_2 (P_1 P_0)^\infty g$ must be invoked to ensure a correct partition.

When the partition is conclusively determined, it is time to convert each class to an individual vertex, which requires choosing a number for it. Numbering the classes c lexicographically by the numbers of the vertices they contain provides for the class containing the vertex numbered 0 to map to 0 and the class containing 1 to map to 1 unless there is no vertex numbered 0, which is allowed.

In that case, the successor of the lexicographic ordinal correctly numbers at least the class containing the vertex numbered 1 if any. An approach to this task by way of a function $(P_3 P_0) g : \mathbb{N} \rightarrow \mathbb{N}$ taking old vertex numbers n directly to new vertex numbers $((P_3 P_0) g) n$ based on such an ordering starts with a definition

$$P_3 = \lambda p. \lambda g. \lambda n. (\{0\} \cup (\mu p) \mathcal{D}(\bigcup g))^\circ p n \quad (6.11)$$

where $(\mu p) \mathcal{D}(\bigcup g) = (\mu P_3) \mathcal{D}(\bigcup g)$ is the set of all sets of vertex numbers by classes, so that the ordinal of the set $p n$ of vertex numbers containing n with respect to it gives most of the result. The remaining term increments it as intended whenever 0 is not a member of any vertex number set. A result $(P_4 P_3 P_0) g$ follows from mapping this function throughout all vertices and adjacency sets in all classes by

$$P_4 = \lambda p. \lambda g. \Pi \bigcup_{(m,e) \in \bigcup g} \{(p g) m\} \times \bigcup_{(t,n) \in e} \{(t, (p g) n)\}$$

to effect the rest of the renumbering.

All that remains for specifying the whole operation is a way of obtaining the initial partition of the graph, which can be inferred from the edge labels alone as noted previously. A graph in the adjacency set representation transformed by $\pi \lambda(m, e). \mathcal{D}(e)$ enables the rest of the partition fusion transformation according to a definition

$$\Phi = (P_4 P_3 P_0) \circ (P_1 P_0)^\infty \circ P_2 \circ (P_1 P_0)^\infty \circ \pi \lambda(m, e). \mathcal{D}(e) \quad (6.12)$$

in terms of P_0 through P_4 and denoted by a fancy symbol in case we want to refer to it again. The final form of the reachability graph for a process $X \in \mathbb{D}$ then follows directly in terms of a function

$$\mathbf{RG} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N}))$$

as the partition fusion of the previous version $\mathbf{RG}_3 X$.

$$\mathbf{RG}(X) = \Phi \mathbf{RG}_3 X \quad (6.13)$$

Wrangling roundup

1. How many cycles are there in each of the graphs in [Figure 6.15](#)?
2. After partition fusion, every vertex in the graph has at most one outgoing edge for each label. True or false?
 - a) If true, what happens to processes capable of non-deterministic behavior?
 - b) If false, what good is it to insist on a class consensus about the destinations of similarly labeled edges?
3. To avoid getting thrown off by all the wild corner cases, describe the reachability graph of a process that is initially
 - a) divergent
 - b) deadlocked
 - c) quiescent but not deadlocked
 - d) not quiescent but not divergent
4. To what function space does each function J_1 through P_4 used in this chapter belong? In other words, what expression comparable to [Equation 6.10](#) corresponds to each of them? Is any of them “polymorphic”? (Functional programmers beware: $f g x$ conventionally means $f(g(x))$, and $f : s \rightarrow t \rightarrow u$ differs from $f : s \times t \rightarrow u$.)
5. How does anonymous edge unification benefit partition fusion?
6. Sketch a hand waving argument by structural induction on process combinator expressions showing that vertex 0 can have no successors in the reachability graph $\mathbf{RG}_2(X)$ of any process $X \in \mathbb{D}$ expressible by process combinators.



In each action we must look beyond the action at our past, present, and future state, and at others whom it affects, and see the relations of all those things. And then we shall be very cautious.

Blaise Pascal

CHAPTER 

TRANSDUCER TUNING

With scarcely time for a breather, the material in this chapter continues the program started in [Chapter 6](#) because the task is not yet complete. The next challenge is to develop the much anticipated transducer representation for a DI process. Compared to the reachability graph, the transducer in its optimal form is more abstract and compact, but its main motivation is its convertibility to a netlist, a trace recognizer, and even back to a simpler canonical form of the original Petri net model.

A netlist is essentially the target representation of the whole design work flow, whose development occupies much of the rest of the book, but the others mentioned above are covered in this chapter along with the transducer model. Trace recognizers are of interest because they allow us to discuss behavioral equivalence and refinement precisely and to verify them automatically. Petri nets are revisited here to effect the promised generalization of the process combinators introduced in [Chapter 5](#) to processes having closed Petri net models, which can occur as intermediate results in expressions invoking the `env` combinator. The construction depends on the ability to transform any transducer to a behaviorally equivalent open Petri net whether it is derived from an open or a closed one. The transformation is somewhat lengthy but otherwise straightforward.

This chapter continues to use some of the notation and conventions introduced in [Chapter 6](#), both standard and idiosyncratic, without much further explanation because they economize the presentation considerably. Readers proceeding non-sequentially through the book are requested to consult [Chapter 6](#) in case of any unfamiliar Greek letters.

7.1 Finite automata

We digress in this section for a quick primer on finite automata and related concepts to benefit a reader who might not be already familiar with them. More knowledgeable readers best proceed even more quickly to [Section 7.2](#) before the newcomer surpasses them.

As if the Petri net, reachability graph, and transducer models considered up to this point were not enough, yet another representation for DI processes is about to be added to the mix, but not without good reason. A finite automaton (plural: “automata”) does one thing well, which is to encode a possibly infinite set of sequences in a form conducive to comparison. While it may be intuitively clear that simulating a transducer and recording the signals it exchanges with its environment could accumulate the set of possible traces, a finite automaton summarizes this information comprehensively and far more elegantly.

7.1.1 Sequences

As noted above, finite automata are concerned with encoding sets of sequences, so it might help to be clear about the concept of a sequence. Sequences, traces, lists, and strings are used interchangeably in this book. While it is possible to treat sequences as fundamental entities like sets and to develop them axiomatically, our present purpose is adequately served by modeling a sequence as a function of a natural variable.

- For a set A and a natural number n , let A^n denote the set of sequences of length n on A .
- Each sequence of length n on A is modeled by a function $s : \{i \in \mathbb{N} \mid i < n\} \rightarrow A$.
- The length of a sequence s is denoted $|s|$, similarly to the cardinality of a set.
- A^0 is allowed and contains a single sequence of length 0 only, denoted ϵ .¹
- A^* is defined as $\bigcup_{n \in \mathbb{N}} A^n$.
- The i -th **item** or **term** of a sequence, $s(i)$, can also be written as s_i .

By the above criteria, any usual way of specifying a function can be used to specify a sequence, but an additional notational device can also be appropriate. If s is a sequence, the expression $a : s$, read “ a cons s ”, is defined as the following sequence.

$$a : s = \lambda i. \begin{cases} a & \text{if } i = 0 \\ s_{i-1} & \text{otherwise} \end{cases} \quad (7.1)$$

Similarly, the **concatenation** of two sequences s and t is denoted as $s \parallel t$, which is defined as follows.

$$s \parallel t = \lambda i. \begin{cases} s_i & \text{if } i < |s| \\ t_{i-|s|} & \text{otherwise} \end{cases} \quad (7.2)$$

It is also convenient to generalize the concept of concatenation to sets of strings X and Y , and to use the same notation for it. For strings s, t and sets X, Y , concatenation is defined as follows.

$$\begin{aligned} X \parallel Y &= (\mu \lambda(s, t). s \parallel t) (X \times Y) \\ s \parallel Y &= \{s\} \parallel Y \\ X \parallel t &= X \parallel \{t\} \end{aligned} \quad (7.3)$$

That is, the concatenation of a pair of sets of strings is the set of all pairwise concatenations of their elements, and the concatenation of a string with a set of strings is the set of concatenations of that string with each element.

¹Technically every alphabet A induces a different zero-length sequence if sequences are defined this way, but this distinction is unlikely to cause confusion and is henceforth ignored.

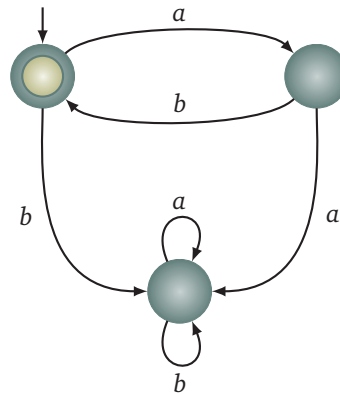


Figure 7.1: a finite automaton accepting any string of alternating a 's and b 's that starts with an a and ends with a b

7.1.2 Bracket notation

Instead of writing a list of three items as $a : b : c : \epsilon$, we may express the same list as $\langle a, b, c \rangle$ using the angle bracket notation. A list containing a single item a can be expressed $\langle a \rangle$. In general, a list x can be expressed by the comma separated sequence of its items enclosed in angle brackets.

$$x = \langle x_0, x_1, x_2 \dots x_{|x|-1} \rangle$$

This notation is a generalization to any number of terms of the notation $\langle a, b \rangle$ used previously. The subscript notation x_i is applicable to a list denoted by a variable x as well as a literal list in angle bracket notation, hence the familiar equivalences $\langle a, b \rangle_0 = a$ and $\langle a, b \rangle_1 = b$.

7.1.3 State graphs

Finite automata can be visualized as graphs of states with labeled directed edges similar to a reachability graph or a transducer, but with some extra features to help them specify a set of strings on an alphabet. One feature is the ability to have specially designated **accepting states**. Another is the requirement of a well defined rule for succession in every state for every member of the alphabet.

Figure 7.1 depicts an example of a finite automaton. It has one accepting state, drawn with an inner circle at the upper left, which is also the initial state. This finite automaton is concerned with strings on an alphabet $\{a, b\}$. Every edge is labeled with one of a or b . Any walk through the graph determines the sequence of a 's and b 's given by the edge labels in the order they are traversed.

A finite automaton specifies a set of strings by a simple rule: if a walk through the graph starts from the initial state and ends in an accepting state, then the sequence of edge labels traversed along the way is in the set. Otherwise, it is not. This rule enables us to test whether any given string is in the set by following the path whose



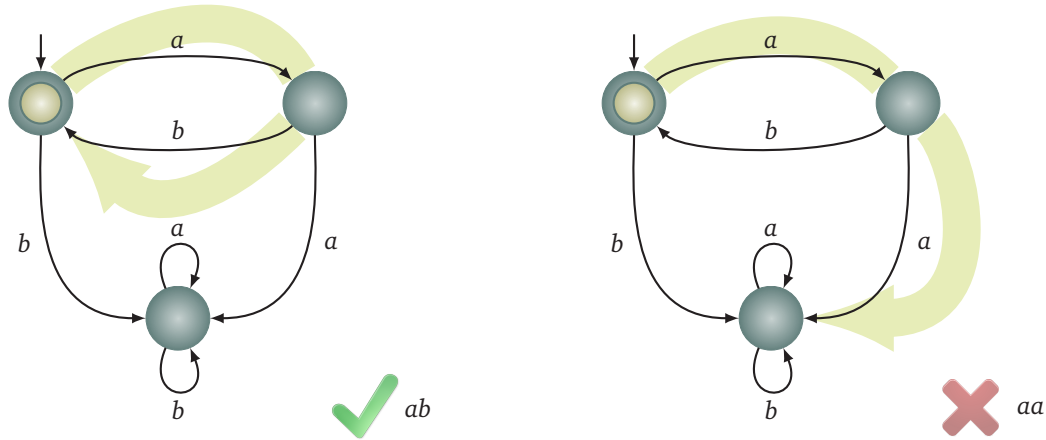


Figure 7.2: The string ab is accepted because a walk traversing edges labeled a and then b ends at an accepting state (left), but the string aa is not accepted (right).

edge labels are spelled out by the string. Trying out this rule for the current example on the strings ab and aa in Figure 7.2 shows that the former is accepted but the latter is not. The automaton also accepts the empty string ϵ , because a walk that starts in the initial state and goes nowhere ends in an accepting state, so the empty string is also in the set.

It is hard not to infer that the finite automaton in Figure 7.1 accepts the set of all strings of the form ϵ , ab , $abab$, $ababab$, and so on. As a result, it gives us an exact finite characterization of an infinite set of strings.

7.1.4 Deterministic finite automata

The formal treatment of finite automata is long established [115, 260], and it differs from the graph representation by adjacency sets used in this book for transducers and reachability graphs. To follow convention, a distinction is needed between *deterministic* and *non-deterministic* finite automata. The DFA is discussed first and the NFA subsequently to the extent they differ.

In its usual presentation, a DFA consists of a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a set of states, Σ is an alphabet, and $\delta : Q \times \Sigma \rightarrow Q$ is a total function taking a current state and an alphabet symbol to a successor state. This function specifies the adjacency relation for the graph. Furthermore, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. These components of a DFA work together to specify the set of strings it accepts, also known as its *language*, according to the formula

$$\mathcal{L}(Q, \Sigma, \delta, q_0, F) = \{s \in \Sigma^* \mid \hat{\delta}(q_0, s) \in F\} \quad (7.4)$$

where $\hat{\delta}$ is the state transition function δ extended in the obvious way to strings $s \in \Sigma^*$.

$$\hat{\delta}(q, s) = \begin{cases} q & \text{if } |s| = 0 \\ (\lambda(a : t). \hat{\delta}(\delta(q, a), t)) s & \text{otherwise} \end{cases} \quad (7.5)$$

7.1.5 Non-deterministic finite automata

The NFA is also packaged as a quintuple $(Q, \Sigma, \delta, q_0, F)$ as above, but differs insofar as the state transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ takes a current state and an optional alphabet symbol to a set of possible successor states rather than to a specific successor. It must be a total function of $Q \times (\Sigma \cup \{\epsilon\})$, but its result may of course be \emptyset for some combinations of states and symbols.

Providing ϵ instead of a member of Σ as an input to δ has the intuitive significance of letting a state change occur without consuming a symbol. A definition of δ using any arbitrary non-member of Σ in place of ϵ would work just as well, but traditionally the empty string ϵ is stipulated despite the confusion it may cause about whether δ operates on symbols or strings.

An extension of the transition function δ analogous to Equation 7.5 for an NFA would be a function $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \rightarrow \mathcal{P}(Q)$ taking a set of possible current states u and a string s to a set of possible future states. An expression for $\hat{\delta}$ is seldom written down explicitly lest it turn out like this,

$$\hat{\delta}(u, s) = \begin{cases} (\rho \lambda q. \delta(q, \epsilon)) u & \text{if } |s| = 0 \\ \hat{\delta}((\rho \lambda q. \delta(q, \epsilon)) u, s) \cup \bigcup_{q \in u} (\lambda(a : t). \hat{\delta}(\delta(q, a), t)) s & \text{otherwise} \end{cases} \quad (7.6)$$

which could have been worse without the percolation operator ρ defined in Equation 6.4. The expression

$$(\rho \lambda q. \delta(q, \epsilon)) u$$

refers to the set of all states reachable through any number of ϵ -transition traversals from any member of u , also known as the ϵ -closure of u .

For a string to be accepted by an NFA requires only that there exist a walk from the initial state to an accepting state with the right edge labels. The sequence of edge labels traversed along the walk must match the string after all occurrences of ϵ are deleted from it. This condition is implicit in Equation 7.6, and allows the language of an NFA to be defined as follows,

$$\mathcal{L}(Q, \Sigma, \delta, q_0, F) = \{s \in \Sigma^* \mid \hat{\delta}(\{q_0\}, s) \cap F \neq \emptyset\}$$

upholding the confusing tradition of using exactly the same notation for the operator \mathcal{L} that maps an NFA to the language it recognizes as we use in the case of a DFA (cf. Equation 7.4).

7.2 The transducer

To recapitulate the description from Section 4.2, the transducer model is also a graph like the reachability graph, but differs from it by associating each edge with a set of concurrent signals rather than just an individual signal or anonymous Petri net transition. This feature makes it more compact than an equivalent reachability graph because the transducer does not require a separate path to express each possible interleaving of concurrent signals. The vertices in a transducer correspond to states of a DI process in a course sense, which may vary from its Petri net markings, so a **state** is a preferable term to a marking for a vertex in a transducer. In this way, the transducer promotes a transactional understanding of the process by specifying sets of acceptable inputs for each state, with a corresponding range of outputs and successor states for each set of inputs.

An example of a reachability graph and an equivalent transducer is shown in Figure 7.3. In the reachability graph at the left, there are two inputs a and b initially enabled. If a happens first, then only b is enabled, and *vice versa*. After both of them happen, two outputs c and d are enabled

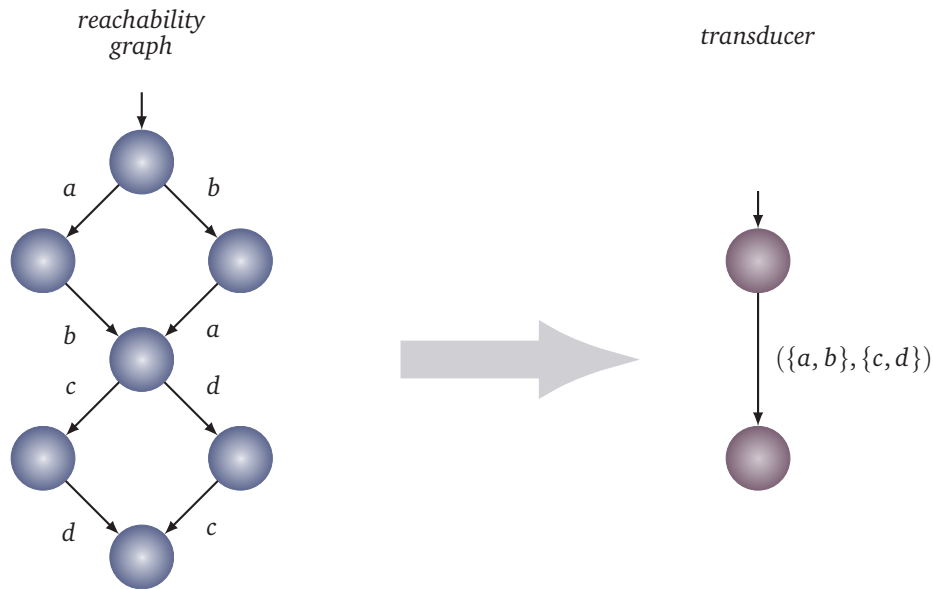


Figure 7.3: A transducer absorbs multiple paths from a reachability graph into a single edge labeled by an i/o burst with inputs a, b and outputs c, d .

in either order. This information is summarized by the equivalent transducer at the right, which requires only a single edge labeled by an ordered pair of an **input burst** $\{a, b\} \subset \mathbb{T}$ and an **output burst** $\{c, d\} \subset \mathbb{T}$, known collectively as an **i/o burst**. The convention is always that all inputs in an input burst must be received by the system before any outputs in the corresponding output burst can be emitted by it.

The rest of this section is concerned with articulating a transformation to construct transducer models automatically.

7.2.1 Overview

Like the reachability graph, the transducer representation of a given process is not unique. A large unnecessarily complicated graph can often be simplified to a smaller semantic equivalent. A good way to achieve the optimum result is to start with a known valid form and then make incremental semantics-preserving improvements. A readily available starting point is always a transducer that is isomorphic to the reachability graph with at most one signal in each i/o burst as shown in the Figure 7.4.

If we overlook temporarily that each i/o burst $(i, o) \in \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})$ expresses the inputs and outputs respectively by separate sets $i, o \in \mathcal{P}(\mathbb{T})$ in favor of a combined set $t = i \cup o \in \mathcal{P}(\mathbb{T})$, then maybe the transducer can be improved by the intuitively appealing rewrite rule illustrated in Figure 7.5. Instead of taking two hops to get from state m to state m'' with separate i/o bursts t and t' , a more parsimonious transducer can do so in a single hop with their union $t \cup t'$ (provided any inputs in t' are safe for m). If this rule is applied to the graph in Figure 7.4 where $t = \{a\}$ and

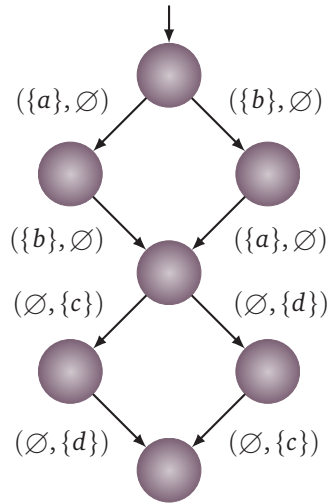


Figure 7.4: A suboptimal transducer equivalent to the one in Figure 7.3 is isomorphic to the reachability graph.

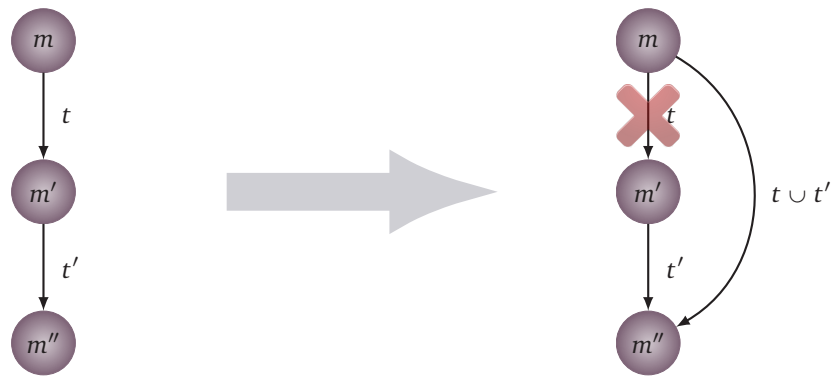


Figure 7.5: the basic rewrite rule for optimizing a transducer

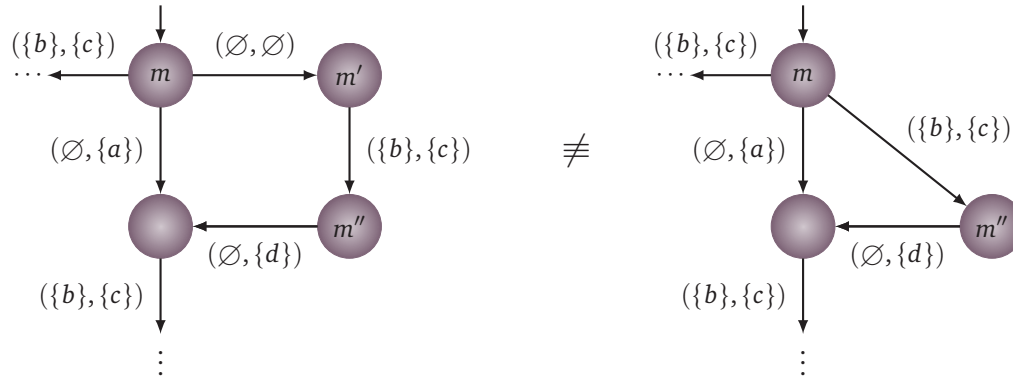


Figure 7.6: Transforming the left transducer to the right would create an output obligation where there was none.

$t' = \{b\}$ are the i/o bursts, then from the outside it still looks as if a and b can happen concurrently, so nothing is changed, and with no edge needed from m to m' anymore, maybe m' can be pruned out of the graph because it is unreachable.

Safely creating edges

Although the rule works well enough in this example, it might not be this easy in general. If both t and t' contain mixtures of inputs and outputs, then putting them together results in an i/o burst $t \cup t'$ that no longer reflects the original semantics. The i/o burst $t \cup t'$ implies that all inputs in both of t and t' must be received by the system first, and only then are all outputs emitted. The original semantics specifies that the outputs in t can start being emitted as soon as all inputs in t are received regardless of those in t' , and that the remaining outputs can be emitted only after the rest of the inputs are received. We can avoid this pitfall by remembering two criteria, which need not be mutually exclusive.

- Any edge from a state m to a state m' together with any edge from m' to m'' whose labels include no inputs induces a new edge from m to m'' labeled by the inputs on the former and their *union of output labels*.
- Almost any edge from a state m to a state m' whose labels include no outputs together with any edge from m' to m'' whose labels include no unsafe inputs for m induces a new edge from m to m'' labeled by the outputs on the latter and their *union of input labels*.

The latter condition needs qualification for the pathological situation depicted in Figure 7.6. With no input from the environment, the transducer on the left may choose to output a , or it may follow the edge labeled (\emptyset, \emptyset) to the quiescent state labeled m' , where it waits forever. The transducer on the right follows a more robust protocol. If the environment never transmits an input, the transducer must output a eventually. If the transducer on the left resulted from a circuit designed that way by mistake, transforming it to the one on the right would allow the mistake to go unnoticed, but this undesirable effect is precisely what follows from combining the incident and outgoing edges on m' based on the latter condition above (along with the pruning of m' subsequently).

In more general terms, the problem with applying this transformation indiscriminately is that it can create an output obligation not mandated by the original semantics, which could result in a possible deadlock being overlooked. To guard against the possibility of creating a spurious output obligation, the latter condition above should be revised as follows.

- Any edge from a state m to a state m' whose labels include no outputs together with any edge from m' to m'' whose labels include no unsafe inputs for m induces a new edge from m to m'' labeled by the outputs on the latter and their *union of input labels* if at least one of these conditions holds.
 - (i) The edge from m to m' is labeled by at least one input.
 - (ii) All other edges from m are labeled by at least one input.
 - (iii) At least one edge from m' is labeled by no inputs.

The intuitive argument for condition (i) is that whether the state m is quiescent or not, attaching yet more input-guarded edges to it changes nothing about its quiescence. The intuition underlying (ii) is that all remaining edges from m are still blocked by inputs even if the path through m' is removed, so there is no opportunity to mandate any hitherto optional output activity. The justification for (iii) is that if the edge from m to m' and some edge from m' onwards are both unlabeled, they preserve any extant unlabeled path from m to a quiescent state as needed to avoid creating an output obligation, but if (i) and (ii) fail and all edges from m' without input labels have output labels, then there is already an output obligation.

Safely deleting edges

Another difficulty is the need for careful consideration about deleting edges. One way of looking at the problem is that an edge can be deleted safely only if it contributes no additional information about the behavior of the process. For example, in Figure 7.5, m' is connected only to m'' , and m'' is reachable anyway from m by an edge with the same labels, so maybe the edge labeled t from m to m' is deletable because it contributes no additional information. However, if there were other edges from m' , then obviously deleting this edge might change things by cutting off the path from m to another state. Even if there were no such path, the edge might have to be retained if m' were state number 0, which represents the final marking in the original process specification. This state must be prevented from becoming unreachable to avoid the loss of relevant semantic information (when it comes to converting the transducer back to a Petri net).

Removing an edge that should have been retained would be a bigger mistake than retaining one that is redundant, so a conservative approach is in order. In the algorithm to be proposed, an edge labeled by an input burst j and arbitrary outputs can be deleted only if there is another edge $((i, o), n)$ labeled by (i, o) from the same state such that i is a proper superset of j . Every path beginning with the edge labeled by j is explored as far as possible along edges whose input labels are within i , and deletable edges must satisfy three further conditions.

- The union of labels along all explored paths precisely matches (i, o) .
- The terminus of the edge to be deleted is not 0.
- The union of all outgoing edges from states at the ends of the explored paths coincides with the union of all outgoing edges from states n connected to the origin by edges labeled (i, o) .

The general approach to deriving the transducer from the reachability graph therefore is to start with an intermediate representation based on the reachability graph as in [Figure 7.4](#), but with only one set of signals for each edge label as in [Figure 7.5](#), then to create all of the new edges at once, then to convert the edge labels to i/o bursts, then to delete the redundant edges, and then to finish with pruning and partition fusion phases.

7.2.2 Derivation

The intermediate representation $g = V_0 \mathbf{RG} X \in \mathcal{P}(\mathbb{N} \times \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N}))$ of the transducer can be derived immediately from the reachability graph by transforming it via the function

$$V_0 : \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times \mathbb{N})) \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N}))$$

defined as follows.

$$V_0 = \lambda g. \bigcup_{(m,e) \in g} \left\{ \left(m, \bigcup_{(t,n) \in e} (\{t\} - \mathbb{V}, n) \right) \right\} \quad (7.7)$$

The result is isomorphic to the reachability graph $\mathbf{RG} X$ in the sense depicted in [Figure 7.4](#), but every anonymous edge in $\mathbf{RG} X$ maps to an edge labeled by the empty set in the result, and every edge labeled by an input or output $t \in \mathbb{T}$ maps to an edge labeled with the singleton set $\{t\}$.

Union of output labels

To derive the transducer, we first add all edges to the graph that can be made by combining any extant edge with an edge that follows from its terminus and is labeled without inputs. That is, for every vertex $(m, e) \in g$, every edge (b, n) in the adjacency set e , and every succeeding edge (o, l) in the adjacency set $(\Psi g) n$, we form a new edge $(b \cup o, l)$ originating from (m, e) , provided that the label o of the succeeding edge is disjoint from I , the input alphabet the process of $X = (I, O, N)$. The transformed graph $(V_1 I) g$ is determined by a function

$$V_1 = \lambda I. \lambda g. \bigcup_{(m,e) \in g} \left\{ \left(m, \left(\rho \lambda (b, n). \bigcup_{(o,l) \in ((\Psi g) n) \cap (\mathcal{P}(\mathbb{T}-I) \times \mathbb{N})} \{ (b \cup o, l) \} \right) e \right) \right\} \quad (7.8)$$

taking the opportunity to percolate this operation exhaustively by [Equation 6.4](#).

Union of input labels

Next we add the edges that can be made by combining an edge labeled without outputs with succeeding edges subject to the conditions mentioned previously. To recapitulate, each edge $(i, n) \in \mathcal{P}(\mathbb{T}) \times \mathbb{N}$ in an adjacency set e of a vertex $(m, e) \in g$ in a transducer g with an input alphabet I leading to a terminus n is eligible to be combined with every succeeding edge $(b, l) \in (\Psi g) n$ not labeled by any unsafe inputs for m

$$b \in \mathcal{P}(\mathbb{T} - (I - \mathcal{D}(e)))$$

if any of three conditions holds. With $p = \mathcal{P}(\mathbb{T} - I)$ temporarily denoting the set of pure output bursts, a candidate edge (i, n) is suitable if its label i contains at least one input

$$i \notin p$$

or if all other edges in e from m are labeled by at least one input

$$(\mathcal{D}(e) - \{i\}) \cap p = \emptyset$$

or if at least one edge in $(\Psi g) n$ from n is labeled by no inputs

$$((\Psi g) n) \cap p \neq \emptyset$$

or in other words if (i, n) is a member of $V_2(\mathcal{P}(\mathbb{T} - I), g, e)$ in terms of a function

$$V_2 : \mathcal{P}(\mathcal{P}(\mathbb{T})) \times \mathcal{P}(\mathbb{N} \times \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N})) \times \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N}) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N})$$

given by

$$V_2 = \lambda(p, g, e). \{(i, n) \in e \mid i \notin p \vee (\mathcal{D}(e) - \{i\}) \cap p = \emptyset \vee \mathcal{D}((\Psi g) n) \cap p \neq \emptyset\}.$$

On this basis we could augment the adjacency set e with the additional edges

$$\bigcup_{(i,n) \in V_2(\mathcal{P}(\mathbb{T}-I), g, e)} \bigcup_{(b,l) \in ((\Psi g) n) \cap (\mathcal{P}(\mathbb{T} - (I - \mathcal{D}(e))) \times \mathbb{N})} \{(i \cup b, l)\}$$

connecting the state m directly to states l , but doing so might create further opportunities to add more edges according to the same criteria, so it is better to rewrite e to the whole set

$$(\lambda d. d \cup \bigcup_{(i,n) \in V_2(\mathcal{P}(\mathbb{T}-I), g, d)} \bigcup_{(b,l) \in ((\Psi g) n) \cap (\mathcal{P}(\mathbb{T} - (I - \mathcal{D}(d))) \times \mathbb{N})} \{(i \cup b, l)\})^\infty e$$

by [Equation 6.3](#). A version of the transducer g with every adjacency set rewritten this way throughout is expressible as $((V_3 V_2) I) g$ in terms of a function

$$V_3 = \lambda v. \lambda I. \lambda g. \Pi \bigcup_{(m,e) \in g} \{m\} \times (\lambda d. d \cup \bigcup_{(i,n) \in v(\mathcal{P}(\mathbb{T}-I), g, d)} \bigcup_{(b,l) \in ((\Psi g) n) \cap (\mathcal{P}(\mathbb{T} - (I - \mathcal{D}(d))) \times \mathbb{N})} \{(i \cup b, l)\})^\infty e$$

and [Equation 6.7](#).

Edge deletion

This result leaves no further need for the intermediate representation $g \in \mathcal{P}(\mathbb{N} \times \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathbb{N}))$ having only a single set of inputs and outputs labeling each edge, so it can be converted to the representation $(V_4 I) g \in \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N}))$ where the i/o burst on each edge consists of separate sets of inputs and outputs with V_4 defined in the obvious way.

$$V_4 = \lambda I. \bigcup_{(m,e) \in g} \{(m, \bigcup_{(b,n) \in e} \{(b \cap I, b - I), n\})\} \quad (7.9)$$

To follow through with the algorithm sketched previously for edge deletion, we envision an input burst $i \in \mathcal{P}(\mathbb{T})$ and an edge $r = ((j, k), l) \in ((\mathcal{P}(i) - \{i\}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N}$ originating from the same state of a transducer g as an edge labeled by i , but whose input burst j is a proper subset of i . A set of edges whose input bursts are contained in i located consecutively along paths beginning with r is expressible as

$$t = (\rho \lambda(b, l). \bigcup_{v \in ((\Psi g) l) \cap ((\mathcal{P}(i) \times \{\emptyset\}) \times \mathbb{N})} \{v\}) \{r\}$$

such that each path stops short of the first non-empty output burst along it. More edges obtained by continuing along the same paths as far as possible without further input are expressible as

$$(\rho \lambda(b, l). \bigcup_{v \in ((\Psi g) l) \cap ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})} \{v\}) t$$

so that the set of all edges along paths beginning with r encompassing at most one input burst followed by at most one output burst is expressible as $(V_5 g) (i, r)$ for V_5 defined as

$$V_5 = \lambda g. \lambda(i, r). (\rho \lambda(b, l). \bigcup_{v \in ((\Psi g) l) \cap ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})} \{v\}) (\rho \lambda(b, l). \bigcup_{v \in ((\Psi g) l) \cap ((\mathcal{P}(i) \times \{\emptyset\}) \times \mathbb{N})} \{r\}). \quad (7.10)$$

To check whether an edge r is safely deletable, it is necessary to check not only the labels on the edges within a set $(V_5 g) (i, r)$, but the effect of traversing one step further. Letting

$$s = \mathcal{R}((V_5 g) (i, r))$$

denote the set of termini of edges in $(V_5 g) (i, r)$, we may write $(V_6 g) s$ with V_6 given by

$$V_6 = \lambda g. \lambda s. \bigcup_{l \in s} ((\Psi g) l) - (\mathcal{D}(\bigcup \mathcal{R}(g)) \times s) \quad (7.11)$$

for the set of edges originating from a state in s and terminating on a state outside of s . In terms of Equation 7.10 and Equation 7.11, one of the necessary conditions noted previously on page 165 for an edge $r = ((j, k), l)$ to be deletable is

$$(V_6 g) \mathcal{R}((V_5 g) (i, r)) = (V_6 g) u$$

where $u \in \mathcal{P}(\mathcal{D}(g))$ is the set of termini n of edges $((i, o), n)$ originating from the same state as r satisfying $j \subset i$. Roughly speaking, this condition means that there is nowhere to go by way of r that can not also be reached by way of the edge $((i, o), n)$. Another one of the three necessary conditions is captured by

$$i \cup o = \bigcup_{(j, k) \in \mathcal{D}((V_5 g) (i, r))} \{j \cup k\}$$

meaning roughly that the observable signals are the same in either alternative. These two conditions along with the requirement for the terminus of deleted edges to be non-zero allow the set of deletable edges from a particular vertex with adjacency set e to be expressible as

$$\bigcup_{((i, o), u) \in \Pi e} \bigcup_{r \in e \cap (((\mathcal{P}(i) - \{i\}) \times \mathcal{P}(\mathbb{T})) \times (\mathbb{N} - \{0\}))} (\lambda a. (\lambda t. \langle \emptyset, \{r\} \rangle_t) \delta_a^{i \cup o} \delta_{(V_6 g) u}^{(V_6 g) \mathcal{R}((V_5 g) (i, r))}) \bigcup_{(j, k) \in \mathcal{D}((V_5 g) (i, r))} \{j \cup k\}$$

and the transducer g with redundant edges deleted from every vertex to be expressible as $V_7 \langle V_5, V_6 \rangle g$ with V_7 defined by

$$V_7 = \lambda v. \lambda g. \Pi \bigcup_{(m, e) \in g} \{m\} \times (e - \bigcup_{((i, o), u) \in \Pi e} \bigcup_{r \in e \cap (((\mathcal{P}(i) - \{i\}) \times \mathcal{P}(\mathbb{T})) \times (\mathbb{N} - \{0\}))} (\lambda a. (\lambda t. \langle \emptyset, \{r\} \rangle_t) \delta_a^{i \cup o} \delta_{(V_1 g) u}^{(V_1 g) \mathcal{R}((V_0 g) (i, r))}) \bigcup_{(j, k) \in \mathcal{D}((V_0 g) (i, r))} \{j \cup k\}).$$

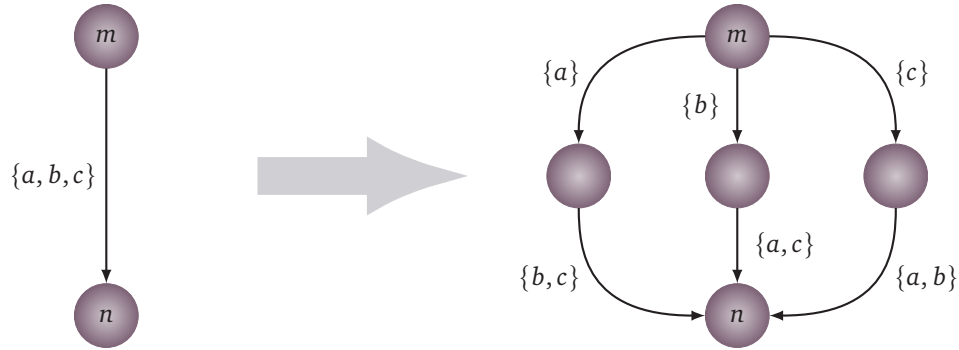


Figure 7.7: Serializing the transducer requires creating a new state for each member of the i/o burst $\{a, b, c\}$. Each created state has an outgoing edge whose i/o burst excludes the corresponding member.

Summary

Two finishing touches on the transducer model are to prune the unreachable vertices using the Γ_1 operator defined by Equation 6.5, and to optimize it by partition fusion as defined by Equation 6.12. The result overall is summarized by a function

$$\mathbf{T} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N}))$$

taking a process $X \in \mathbb{D}$ to its transducer model $\mathbf{T}(X)$ according to this definition.

$$\mathbf{T}(X) = (\lambda(I, O, N). \Phi \Gamma_1 V_7 \langle V_5, V_6 \rangle (V_4 I) ((V_3 V_2) I) (V_1 I) V_0 \mathbf{RG} X) X \quad (7.12)$$

The definitions of the operators Γ_1 and Φ on reachability graphs fortunately require no modification for transducers.

7.3 Serial transducers

Taking the trouble to aggregate the signals into concurrent bursts as much as possible in the previous section has been an important step toward the upcoming construction of open Petri net models in Section 7.5, but before that we have to do the opposite by teasing the bursts apart. This job needs to be done as a means of deriving the trace recognizing automata in Section 7.4, and the netlist eventually. The resulting transducer has at most one signal labeling each edge, much like the depiction in Figure 7.4 but may be an improvement on it because it benefits from partition fusion performed on a more abstract representation.

7.3.1 Overview

The transformation to a serial transducer is built on the rewrite rule illustrated in Figure 7.7. It is convenient here as well to use an intermediate representation temporarily featuring i/o bursts $b \in \mathcal{P}(\mathbb{T})$ that do not distinguish between inputs and outputs. The idea is to transform any edge labeled b between a state m and a state n with $|b| > 1$ to a set of $|b|$ edges each pointing from m to

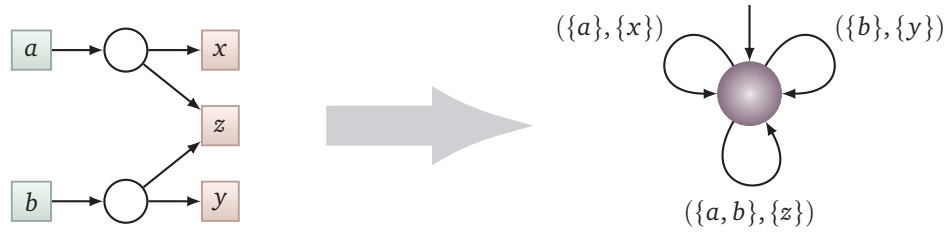


Figure 7.8: a transducer with one input burst containing another

a newly created vertex. The edge from m to each new vertex is labeled by a set containing a distinct member of b , and the edge from that vertex to n is labeled by the rest of the members of b . The serial transducer is obtained for the most part by applying this rule repeatedly throughout the graph until all i/o bursts are worn down to at most one member.

The rewrite rule might not always be as easy to apply as shown in Figure 7.7 if the i/o burst b contains both inputs and outputs. Then we must deal with the inputs first and postpone dealing with the outputs. Otherwise, it might be possible for outputs to get ahead of inputs they are meant to follow along the created path. Following this plan requires inspecting b to see if it intersects I , the input alphabet of the process. If it does, then only edges labeled by members of $b \cap I$ should be created. If b contains no inputs, then the rule applies uniformly to all of its members.

Another difficulty with this rewrite rule is the assignment of unique state numbers to the newly created vertices so as not to clash with any state numbers in the given transducer g already. Potentially any state m connected to a state n by an edge labeled b could spawn a whole hypercube of up to $2^{|b|} - 2$ intervening states. It might be worth planning in advance to reserve that many state numbers for every combination of states and i/o bursts.

In any case, the above considerations are minor compared to one that becomes apparent from looking ahead to the matter of deriving a trace recognizer. According to the method sketched informally in Section 4.4, the end game is to identify accepting states in the trace recognizer with quiescent states in the transducer, which are recognizable as such when their outgoing edge labels contain only input signals. However, this decision procedure is not always adequate.

It is easy to construct an example of a process specification that isolates the issue. Nothing prevents a process from being specified as

$$P = \text{loop} (\text{f} \widetilde{\text{alt}}) \widetilde{\text{seq}}^* \langle (\text{get } a, \text{put } x), (\text{get } b, \text{put } y), (\widetilde{\text{par}} (\text{get } a, \text{get } b), \text{put } z) \rangle \quad (7.13)$$

leading to a transducer model in which one input burst is a proper subset of another associated with the same state as shown in Figure 7.8.² The protocol mandated by this specification requires the process to acknowledge an input of a with an output of x , an input of b with an output of y , and concurrent inputs of a and b with an output of z . When the process receives concurrent or nearly concurrent inputs, it may of course choose to treat the inputs as concurrent or sequential, because there is no enforceable alternative consistent with delay insensitivity. However, having received a single input, the process does not get to choose whether to wait for the next one. If the environment never sends another input, the process must nevertheless transmit an output.

²The illustrated Petri net results from local optimizations described in Chapter 9.

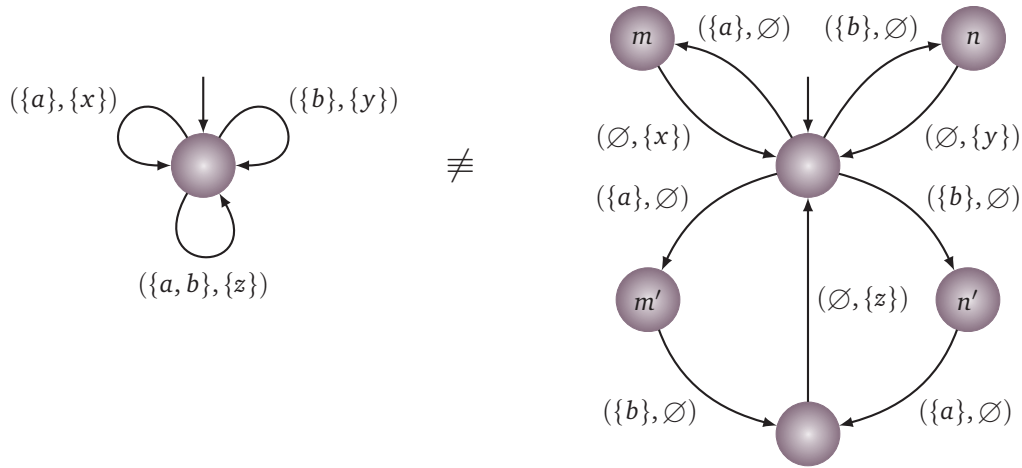


Figure 7.9: The traces $\langle a \rangle$ and $\langle b \rangle$ are quiescent for one transducer and not the other.

Figure 7.9 shows what happens when we serialize this transducer by transforming each edge to an ensemble of intermediate states, with the original version on the left and the serialized version on the right. The path in the serialized version from the initial state to m' can be taken upon receipt of an input a , whereupon no progress is possible without an input of b . Regardless of the availability of a path to m , this condition is sufficient for the trace $\langle a \rangle$ to be quiescent, contrary to the original specification. Similar reasoning applies to $\langle b \rangle$.

At this point we could go back to the drawing board and demand a more sophisticated serializing transformation that preserves behavioral equivalence, and indeed we revisit this question in Chapter 15 on state based circuit synthesis when it is not so easily avoided. However, if the serial transducer is regarded only as a stepping stone from the transducer $\mathbf{T}X$ to a trace recognizing automaton, then a simple workaround is to exclude m' and n' from the set of accepting states. The combined solution can be a pair

$$\mathbf{ST}X = (g, S)$$

where g is a serial transducer and $S \subseteq \mathcal{D}(g)$ is an explicit record of its quiescent states retained during the course of the transformation. In the current example $S = \{1\}$ should contain only the initial state.

To be included in the set S of quiescent states without misrepresenting the semantics of the original process specification, a state of the serial transducer must meet two necessary conditions. The first is that all of its outgoing edges must be labeled by at least one input signal, as usual. The second applies additionally to states that are created for the serial transducer along a path between two states q_0 and q_1 of the original. The intermediate states are counted as quiescent only if it is also true that the union of input bursts labeling the edges along the path from the most recent original state q_0 does not coincide with any complete input burst labeling an outgoing edge from q_0 in the original transducer. This condition would exclude m' in the current example because the path from the initial state to m' is labeled by the set $\{a\}$, which is the whole input burst along an outgoing edge from the initial state (to itself) in the original transducer at the left of Figure 7.9. This condition never affects transducers in which no input burst is a proper subset of any other.

Keeping track of the information needed to evaluate these conditions calls for another temporary alteration to the representation of a transducer. Not only are the i/o bursts $(i, o) \in \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})$ reduced to single sets $b = i \cup o \in \mathcal{P}(\mathbb{T})$ as noted above, but each transducer state $s \in \mathcal{D}(g)$ is represented as a pair

$$s = (q, w) = (\langle q_0, q_1 \rangle, \langle w_0, w_1 \rangle) \in \mathbb{N}^2 \times \mathcal{P}(\mathbb{T})^2$$

with q_0 and q_1 interpreted as above, w_0 containing the input signals needed to change the state from q_0 to s , and w_1 containing the input or output signals needed to change the state the rest of the way from s to q_1 . Because the set of pairs (q, w) is countable when w is restricted to any particular finite alphabet, there is no obstacle to converting later from this representation to states encoded by natural numbers.

7.3.2 Derivation

The preceding overview should serve to motivate the more detailed elaboration of the serial transducer transformation to follow in this section, now taken in several steps due to its length.

Adjacency sets

Expressing the ideas in [Section 7.3.1](#) more precisely starts with an observation that any intermediate state represented by a pair (q, w) as interpreted above immediately determines its whole adjacency set. Each outgoing edge from such a state is labeled by a signal $j \in w_1$ and points to a state

$$(q, \langle w_0 \cup (I \cap \{j\}), w_1 - \{j\} \rangle)$$

with one less signal ahead of it and maybe one more behind it on the path from q_0 to q_1 . The term w_0 need only record input signals, so j is added to it only if j is a member of I , the input alphabet of the process. To maintain the condition that input bursts must be received in full before the corresponding outputs are emitted, the edge

$$(\{j\}, (q, \langle w_0 \cup (I \cap \{j\}), w_1 - \{j\} \rangle))$$

can be present for outputs j only if outputs are the only members left in w_1 , and is restricted to inputs otherwise. The whole adjacency set in either case is given by $(V_8 I) (q, w)$ in terms of a function

$$V_8 : \mathcal{P}(\mathbb{T}) \rightarrow ((\mathbb{N}^2 \times \mathcal{P}(\mathbb{T})^2) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{T}) \times (\mathbb{N}^2 \times \mathcal{P}(\mathbb{T})^2)))$$

defined as

$$V_8 = \lambda I. \lambda (q, w). \bigcup_{j \in (\lambda k. \langle w_1 \cap I, w_1 \rangle_k) \delta_{\emptyset}^{w_1 \cap I}} \{(\{j\}, (q, \langle w_0 \cup (I \cap \{j\}), w_1 - \{j\} \rangle))\}. \quad (7.14)$$

Origins and termini

Whereas this function would appear to generate all edges needed to populate the serial transducer along any path through intermediate states, further provisions are needed for the states at the beginnings and ends of these paths, which correspond to the states in the original transducer g prior to the transformation. Rewriting each outgoing edge $((i, o), n) \in e$ from a vertex $(m, e) \in g$ to a set of edges $(V_8 I) (\langle m, n \rangle, \langle \emptyset, i \cup o \rangle)$ originating at whatever state (q, w) we choose to identify

with m accounts for first edge in each path. The last edge in each path lands on the state identified with n without modification to Equation 7.14 if we identify all pairs of the form

$$\langle\langle q_0, n \rangle, \langle w_0, \emptyset \rangle\rangle$$

with the state numbered n in the original transducer. The arbitrary choice of

$$\langle\langle 0, m \rangle, \langle \emptyset, \emptyset \rangle\rangle$$

as the representation of the state m beginning the path is consistent with this constraint. A subset of the serial transducer representation including only the vertices identified with those in g therefore is expressible as

$$\prod \prod_{(m,e) \in g} \{ \langle\langle 0, m \rangle, \langle \emptyset, \emptyset \rangle \rangle \} \times \prod_{((i,o),n) \in e} (V_8 I) \langle\langle m, n \rangle, \langle \emptyset, i \cup o \rangle \rangle.$$

Filling in the rest of the edges and the intermediate states is a matter of percolating from this initial form to $g' = (V_9 V_8 I) g$ for V_9 given by

$$V_9 = \lambda v. \lambda g. (\rho \lambda (s, a). \prod_{(b,t) \in a} \{ \langle\langle t, v \ t \rangle \rangle \}) \prod \prod_{(m,e) \in g} \{ \langle\langle 0, m \rangle, \langle \emptyset, \emptyset \rangle \rangle \} \times \prod_{((i,o),n) \in e} v \langle\langle m, n \rangle, \langle \emptyset, i \cup o \rangle \rangle.$$

State numbering

Converting states (q, w) in the intermediate representation of the serial transducer to natural numbers in the final result can be done by any choice of function from $\mathbb{N}^2 \times \mathcal{P}(\mathbb{T})^2$ to \mathbb{N} that maps arguments of the form $\langle\langle q_0, n \rangle, \langle w_0, \emptyset \rangle \rangle$ to n as noted above, but that otherwise yields a distinct value different from any state $m \in \mathcal{D}(g)$ of the original transducer. For concreteness, let $\eta : \mathbb{T} \rightarrow \mathbb{N}$ denote an arbitrary fixed injective function defined at least for members of the process alphabet as proposed in Section 5.2.3. Then the set

$$\mathcal{D}(g)^2 \times \left((\mu^2 \eta) \prod_{b \in \mathcal{D}(\mathcal{D}(\cup \mathcal{R}(g))) \cup \mathcal{R}(\mathcal{D}(\cup \mathcal{R}(g)))} \mathcal{P}(b) \right)^2 \subset \mathbb{N}^2 \times \mathcal{P}(\mathbb{N})^2$$

of all combinations of states and i/o burst images possible to associate with the transducer g is totally ordered according to conventions specified in Section 6.1.3, procuring existence and uniqueness of an ordinal

$$\left(\mathcal{D}(g)^2 \times \left((\mu^2 \eta) \prod_{b \in \mathcal{D}(\mathcal{D}(\cup \mathcal{R}(g))) \cup \mathcal{R}(\mathcal{D}(\cup \mathcal{R}(g)))} \mathcal{P}(b) \right)^2 \right)^\circ (q, (\mu \eta)^* w) \in \mathbb{N}$$

for any state

$$(q, w) \in \mathcal{D}(g)^2 \times \mathcal{P}(\mathcal{D}(\mathcal{D}(\cup \mathcal{R}(g))) \cup \mathcal{R}(\mathcal{D}(\cup \mathcal{R}(g))))^2.$$

To ensure a result that does not clash with any state $m \in \mathcal{D}(g)$, we can safely offset this value to

$$(\max \mathcal{D}(g)) + \left(\mathcal{D}(g)^2 \times \left((\mu^2 \eta) \prod_{b \in \mathcal{D}(\mathcal{D}(\cup \mathcal{R}(g))) \cup \mathcal{R}(\mathcal{D}(\cup \mathcal{R}(g)))} \mathcal{P}(b) \right)^2 \right)^\circ (q, (\mu \eta)^* w)$$

for all $(q, w) \neq \langle\langle 0, 0 \rangle, \langle \emptyset, \emptyset \rangle \rangle$ and ensure the last condition in $(V_{10} g) (q, w)$ for V_{10} defined by

$$V_{10} = \lambda g. \lambda (q, w). \langle (\max \mathcal{D}(g)) + \left(\mathcal{D}(g)^2 \times \left((\mu^2 \eta) \prod_{b \in \mathcal{D}(\mathcal{D}(\cup \mathcal{R}(g))) \cup \mathcal{R}(\mathcal{D}(\cup \mathcal{R}(g)))} \mathcal{P}(b) \right)^2 \right)^\circ (q, (\mu \eta)^* w), q_1 \rangle_{\delta_{\emptyset}^{w_1}}. \quad (7.15)$$

Quiescent states

The quiescent states $(q, w) \in \mathcal{D}(g')$ of the serial transducer g' according to the criteria described in [Section 7.3.1](#) are restricted to vertices $((q, w), r) \in g'$ with adjacency sets

$$r \in \mathcal{P}((\mathcal{P}(I) - \{\emptyset\}) \times \mathcal{D}(g'))$$

wherein every edge label contains a symbol from I , the input alphabet of the process, and also restricted to states with w_0 absent from the set of input bursts

$$\bigcup_{((i,o),n) \in (\Psi g)_0} \{i\}$$

relative to the original (not serialized) transducer g . The set of quiescent states is expressible more succinctly as

$$\bigcup_{((q,w),r) \in g' \cap (\mathcal{D}(g') \times \mathcal{P}((\mathcal{P}(I) - \{\emptyset\}) \times \mathcal{D}(g')))} (\lambda k. \langle \{(q, w)\}, \emptyset \rangle_k) \delta_{\emptyset}^{\{w_0\} - f q_0} \in \mathcal{P}(\mathbb{N}^2 \times \mathcal{P}(\mathbb{T})^2)$$

in terms of a function

$$f = \lambda m. \bigcup_{((i,o),n) \in (\Psi g)_m} \{i\}$$

with respect to g , or we can take the opportunity to convert them to natural numbers by

$$\bigcup_{((q,w),r) \in g' \cap (\mathcal{D}(g') \times \mathcal{P}((\mathcal{P}(I) - \{\emptyset\}) \times \mathcal{D}(g')))} (\lambda k. \langle \{(V_{10} g) (q, w)\}, \emptyset \rangle_k) \delta_{\emptyset}^{\{w_0\} - f q_0} \in \mathcal{P}(\mathbb{N}).$$

Preferred representation

Converting the serial transducer to the preferred form of numerical states and pairs of edge labels is a straightforward matter of writing

$$\bigcup_{(s,a) \in g'} \{((V_{10} g) s, \bigcup_{(b,t) \in a} \{((b \cap I, b - I), (V_{10} g) t)\})\}$$

with g' and I as above. To pair the serial transducer in its final form with the set of quiescent states as planned, we may write

$$((V_{11} I) \langle V_{10} g, \lambda m. \bigcup_{((i,o),n) \in (\Psi g)_m} \{i\} \rangle) g'$$

for V_{11} given by

$$V_{11} = \lambda I. \lambda v. \lambda g'. \left(\bigcup_{(s,a) \in g'} \{v_0 s, \bigcup_{(b,t) \in a} \{((b \cap I, b - I), v_0 t)\}\}, \bigcup_{((q,w),r) \in g' \cap (\mathcal{D}(g') \times \mathcal{P}((\mathcal{P}(I) - \{\emptyset\}) \times \mathcal{D}(g')))} (\lambda k. \langle \{v_0(q, w)\}, \emptyset \rangle_k) \delta_{\emptyset}^{\{w_0\} - v_1 q_0} \right).$$

We may note in passing that this result as written would be prone to an exponentially sparse state space, but the state numbers could be transformed to a consecutive set if desired by substituting a function

$$v'_0 = (\{0\} \cup (\mu v_0) \mathcal{D}(g'))^\circ \circ v_0$$

for v_0 in this expression without affecting the semantics (*cf.* [Equation 6.11](#)).

Overall transformation

Attentive readers will have noted that $V_{10} g$ is undefined according to Equation 7.15 when the transducer g is empty, as in the case of an initially divergent process (cf. item 3, page 155). To cover this case while also taking the opportunity to express the serial transducer in terms of a process specification $X \in \mathbb{D}$, let

$$\mathbf{ST} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})) \times \mathcal{P}(\mathbb{N})$$

be defined as

$$\mathbf{ST}(X) = (\lambda(I, O, N). (\lambda g. \langle (V_{11} I) \langle V_{10} g, \lambda m. \bigcup_{((i,o),n) \in (\Psi g) m} \{i\} \rangle (V_9 V_8 I) g, (g, \emptyset) \rangle_{\delta_g} \mathbf{T} X) X. \quad (7.16)$$

7.4 Trace recognizers

Trace recognizing automata settle questions of equivalence and compatibility between DI circuits, making them essential for quality control in any sensible engineering work flow. A formal specification of the refinement relational trace recognizer and the concepts of behavioral equivalence and refinement are the subjects of this section. These specifications rely on the serial transducer model developed in Section 7.3. See Section 4.3 for more motivation.

7.4.1 Non-deterministic relational trace recognizer

The relational trace recognizer for a DI process $X \in \mathbb{D}$ is the first order of business. We start with the NFA for this purpose because it is easier than a DFA based on the serial transducer model. An equivalent DFA version derived from this one follows in Section 7.4.2.

States and symbols

It may be recalled from Section 4.3.4 that the relational trace set includes all quiescent traces and all divergent traces, where a quiescent trace is defined as one for which no progress is possible at its conclusion until the environment provides another input. A divergent trace follows from any prohibited input, and any trace with a divergent prefix is also divergent. An automaton that recognizes this set for a process $X = (I, O, N) \in \mathbb{D}$ therefore can have the same alphabet

$$\Sigma = I \cup O$$

and the same state space as the serial transducer g in $(g, S) = \mathbf{ST} X$ with the addition of one trap state to serve as the destination of divergent traces. For a non-empty graph g , a state space

$$Q = \mathcal{D}(g) \cup \{1 + \max \mathcal{D}(g)\}$$

suffices for the NFA, where $1 + \max \mathcal{D}(g)$ is the number of the trap state. The initial state number

$$q_0 = 1$$

is inherited from the serial transducer via the transducer and the reachability graph.

Accepting states

As for the set of accepting states F , the trap state $1 + \max \mathcal{D}(g)$ must be a member because the automaton accepts divergent traces. Quiescent states must also be members of F , because quiescent traces are also accepted. The set of quiescent states is given already as S in $(g, S) = \mathbf{ST} X$, so we have

$$F = (U_0 S) g$$

in terms of a function U_0 defined as follows.

$$U_0 = \lambda S. \lambda g. S \cup \{1 + \max \mathcal{D}(g)\}$$

Transition function

We have now covered everything needed for an automaton $(Q, \Sigma, \delta, q_0, F)$ except the transition function δ . In addition to simulating the adjacency relation of the serial transducer, the transition function must send the system to the trap state $1 + \max \mathcal{D}(g)$ in the event of an unspecified input and always keep it there subsequently.

Simulation To address the simulation aspect first, a state $q \in \mathcal{D}(g)$ determines an adjacency set $(\Psi g) q$ containing edges of the form $((i, o), m')$. As far as the automaton is concerned, the set of successor states $\delta(q, t)$ from state q for a signal or epsilon transition $t \in I \cup O \cup \{\epsilon\}$ should contain precisely those states m' for which either t is the signal in the corresponding i/o burst $i \cup o$, or for which $i \cup o$ is empty if t is ϵ .

$$t \in i \cup o \vee (t = \epsilon \wedge i \cup o = \emptyset)$$

A function $U_1 g$ that takes a pair (q, t) to this set of states

$$U_1 = \lambda g. \lambda(q, t). \mathcal{R}(\{((i, o), m') \in (\Psi g) q \mid t \in i \cup o \vee (t = \epsilon \wedge i \cup o = \emptyset)\})$$

simulates the serial transducer for valid combinations of signals and states other than the trap state $1 + \max \mathcal{D}(g)$ but yields the empty set otherwise.

Trapping Extending the transition function to take account of the trap state is straightforward in the form of a function $(U_2 u) g$, where $u = U_1$ is defined above, and U_2 is defined as follows.

$$U_2 = \lambda u. \lambda g. \lambda(q, t). (\lambda i. \langle (u g) (q, t), \{1 + \max \mathcal{D}(g)\} \rangle_i) \delta_q^{1 + \max \mathcal{D}(g)}$$

By this definition, if q is the trap state $1 + \max \mathcal{D}(g)$, then $(U_2 g) (q, t)$ is $\{1 + \max \mathcal{D}(g)\}$ regardless of t , but is otherwise identical to $(u g) (q, t)$.

Divergence The remaining requirement of the transition function mentioned above is to ensure the acceptance of divergent traces. It suffices for it to yield the trap state for any input signal $t \in I$ and any state q for which the serial transducer shows no outgoing edge from q labeled by t . This condition is met when the set of states $d = (u g) (q, t)$ is empty with $u = U_2 U_1$ as defined above and t is an input. A set equal to $\{1 + \max \mathcal{D}(g)\}$ in this case but d otherwise

$$(\lambda i. \langle d, \{1 + \max \mathcal{D}(g)\} \rangle_i) \delta_d^\emptyset \delta_I^{I \cup \{t\}}$$

suggests a complete transition function $((U_3 I) u) g$ with U_3 given by

$$U_3 = \lambda I. \lambda u. \lambda g. (\lambda d. (\lambda i. \langle d, \{1 + \max \mathcal{D}(g)\} \rangle_i) \delta_d^\emptyset \delta_I^{I \cup \{t\}}) \circ (u g).$$

Summary

While on the subject of divergence, we should also attend to the case of an initially divergent process, whose reachability graph and serial transducer are therefore empty (cf. [item 3](#), page 155), especially because the derivation up to this point assumes a non-empty transducer g for the trap state number expressed as $1 + \max \mathcal{D}(g)$ to be well defined. The relational trace set of a divergent process with an alphabet $a = I \cup O$ contains every string in a^* , so its non-deterministic relational trace recognizer could be

$$(Q, \Sigma, \delta, q_0, F) = (\{1\}, a, \lambda(q, t). \{1\}, 1, \{1\})$$

with only a single state 1, which is also its initial state, the only accepting state, and the only possible output of its transition function. An expression $((U_4 a) \langle U_0 S, (U_3 I) U_2 U_1 \rangle) g$ covering either case follows from a definition of U_4 given by

$$U_4 = \lambda a. \lambda u. \lambda g. \langle (\mathcal{D}(g) \cup \{1 + \max \mathcal{D}(g)\}, a, u_1 g, 1, u_0 g), (\{1\}, a, \lambda(q, t). \{1\}, 1, \{1\}) \rangle_{\delta_g^{\emptyset}}$$

This definition enables a summary of the non-deterministic relational trace recognizer for any process $X \in \mathbb{D}$ expressible as $\mathbf{NR}(X)$ in terms of the serial transducer $\mathbf{ST} X$ and a function defined as follows.

$$\mathbf{NR}(X) = (\lambda(I, O, N). (\lambda(g, S). (U_4(I \cup O) \langle U_0 S, (U_3 I) U_2 U_1 \rangle) g) \mathbf{ST} X) X$$

7.4.2 Deterministic relational trace recognizer

There is no need to look further than the NFA for testing behavioral equivalence or refinement between DI processes as described in [Section 7.4.3](#), because this representation is most conducive to an efficient decision procedure for language inclusion [1]. However, a DFA may be of interest as an aid to intuition or may simplify *ad hoc* investigations of trace set properties on small examples (e.g., as in [Section 4.4.4](#)).

The standard way of converting an NFA to an equivalent DFA is the **power set construction**, whereby each state of the DFA is identified with a set of states in the NFA. In the worst case, the state space of the DFA can be exponentially larger than that of the NFA. In typical cases it is larger but not unmanageable. Details of this construction along with pruning and partition fusion optimizations are noted briefly in the remainder of this section.

Power sets

Formally it is straightforward to express the DFA equivalent d to an NFA n as $d = U_5 n$ in terms of a function U_5 defined by

$$U_5 = \lambda(Q, \Sigma, \delta, q_0, F). (\mathcal{P}(Q), \Sigma, \lambda(u, t). \bigcup_{q \in (\rho \lambda s. \delta(s, \epsilon)) u} \delta(q, t), \{q_0\}, \mathcal{P}(Q) - \mathcal{P}(Q - F)).$$

That is, the state space of d is the power set of that of n , the alphabets are the same, the initial state of d is the singleton set of the initial state of n , and the set of accepting states of d is the set of all sets of states of n in which any member is an accepting state of n .

The transition function is the only complicated part, operating on a DFA state u , hence a set of NFA states, and a signal t . The transition function expresses a connection from a state u of the DFA by an edge labeled t to whatever any member of $q \in u$ in the NFA would reach by edges with

that label. Generally there are multiple successor states $\delta(q, t)$ in the NFA, so the result of the DFA transition function is given by their union, which becomes a unique state in the DFA. To allow for ϵ -transitions, the union of $\delta(q, t)$ is taken not over all $q \in u$, but over the ϵ -closure of u . (See page 161.)

Reachable states

Many members of the state space $\mathcal{P}(Q)$ according to U_5 n may be unreachable and therefore not worth enumerating. A better definition of $d = U_6 U_5 n$ restricts the DFA to reachable states by percolating the transition function starting from the initial state as shown.

$$U_6 = \lambda(Q, \Sigma, \delta, q, F). (\lambda Q'. (Q', \Sigma, \delta, q, F \cap Q')) (\rho \lambda s. (\mu \delta) (\{s\} \times \Sigma)) \{q\}$$

Partition fusion

The DFA can be improved further by a state minimization algorithm similar to the one discussed in Section 6.6 but simpler where state machines are concerned following [115] and relying on notation introduced in Equation 6.3 and Equation 6.7. According to this algorithm, the state space Q of a DFA $(Q, \Sigma, \delta, q_0, F)$ is partitioned initially into accepting and non-accepting states

$$p = \{F, Q - F\}$$

and thereafter every class $c \in p$ is partitioned further into subclasses as needed to ensure that all states $q \in c$ share a common value of

$$\bigcup_{t \in \Sigma} \{t\} \times \{d \in p \mid \delta(q, t) \in d\}$$

which is the set of all pairs (t, d) of destination classes d connected to q by edges labeled with the alphabet symbol t according to the transition function δ , to yield a partition

$$r = (\lambda p. \bigcup_{c \in p} (\pi \lambda q. \bigcup_{t \in \Sigma} \{t\} \times \{d \in p \mid \delta(q, t) \in d\}) c)^\infty \{F, Q - F\}.$$

Each class $c \in r$ corresponds to a state in a reduced state space, and a function

$$\lambda q. r^\circ \bigcup \{c \in r \mid q \in c\}$$

transforms any state $q \in Q$ from the original state space to its equivalent in the new state space, which is the ordinal relative to r of the class $c \in r$ containing q . To summarize up to this point, we can express a state conversion function $f = U_7(\Sigma, \delta) \{F, Q - F\} : Q \rightarrow \mathbb{N}$ by

$$U_7 = \lambda(\Sigma, \delta). (\lambda r. \lambda q. r^\circ \bigcup \{c \in r \mid q \in c\}) \circ (\lambda p. \bigcup_{c \in p} (\pi \lambda q. \bigcup_{t \in \Sigma} \{t\} \times \{d \in p \mid \delta(q, t) \in d\}) c)^\infty.$$

The rest is only a matter of transforming the given DFA accordingly by mapping f over the sets of states Q and F , rewriting the initial state q as $f q$, and replacing the transition function δ with

$$\lambda(u, t). f \min \bigcup_{e \in \{s \in Q \mid u = fs\} \times \{t\}} \{\delta e\}$$

which defines a function of a new state u and an alphabet symbol t as the new state corresponding an arbitrary result obtained by the original transition function δ from any pair $e = (s, t)$ containing an original state s corresponding to u . Denoting this transformation $U_8 U_7$ in terms of a function U_8 defined by

$$U_8 = \lambda h. \lambda(Q, \Sigma, \delta, q, F). (\lambda f. ((\mu f) Q, \Sigma, \lambda(u, t). f \min \bigcup_{e \in \{s \in Q \mid u=fs\} \times \{t\}} \{\delta e\}, f q, (\mu f) F)) h(\Sigma, \delta) \{F, Q - F\}$$

results in an optimal deterministic relational trace recognizer for a process $X \in \mathbb{D}$ as $\mathbf{RR}(X)$ given by

$$\mathbf{RR}(X) = (U_8 U_7) U_6 U_5 \mathbf{NR} X \quad (7.17)$$

in terms of the non-deterministic relational trace recognizer $\mathbf{NR} X$.

7.4.3 Behavioral equivalence

The relational trace set of a process is important enough to deserve a fancy notation because it embodies a great deal of information about the process and what can be done with it. The relational trace set of a process X hereafter is denoted

$$\llbracket X \rrbracket = \mathcal{L} \mathbf{RR}(X)$$

using the \mathcal{L} operator defined in Equation 7.4 and the deterministic relational trace recognizer defined in Equation 7.17. Brackets like these are used customarily in the literature of denotational semantics to enclose arbitrary verbiage whose “meaning” is of interest, which is to say its image in some designated (ideally more abstract) semantic domain.

The concept of behavioral equivalence now yields easily to a formal specification implying a computable decision procedure. Two processes are behaviorally equivalent to each other when they are not distinguishable by any inputs or outputs they might exchange with an environment. For processes $X = (I_X, O_X, N_X) \in \mathbb{D}$ and $Y = (I_Y, O_Y, N_Y) \in \mathbb{D}$, the behavioral equivalence relation $X \equiv Y$ is defined as that which satisfies

$$X \equiv Y \Leftrightarrow I_X = I_Y \wedge O_X = O_Y \wedge \llbracket X \rrbracket = \llbracket Y \rrbracket.$$

Refinement is an even more useful concept than behavioral equivalence, and also easily expressible in these terms. A process Y refines a process X when Y is a compatible replacement for X in any environment. Compatibility implies that Y can not deadlock or diverge in any circumstance where X would not, but is unconstrained where X diverges. This relation is defined as follows.

$$X \sqsubseteq Y \Leftrightarrow I_X = I_Y \wedge O_X = O_Y \wedge \llbracket X \rrbracket \supseteq \llbracket Y \rrbracket \quad (7.18)$$

Note that the relational operators are oppositely directed. That is, X precedes Y in the refinement ordering if the relational trace set of X is a *superset* of that of Y . This ordering is consistent with the least refined process being the most unpredictable. As such, it is capable of participating in any trace in $(I_X \cup O_X)^*$ with the environment. See Section 4.3.3 for further discussion and motivation.

7.4.4 Alternative extensional descriptions

However useful it may be, the relational trace set is built from the union of two sets with more readily intuitive descriptions: the quiescent trace set, containing the traces whereupon the process

waits for further input before proceeding, and the divergent trace set, whose members render the process unpredictable due to a prohibited input. If an automaton $\mathbf{QR}(X)$ and an automaton $\mathbf{DR}(X)$ respectively recognizing the quiescent and divergent traces of X were known, then another way of defining the relational trace set $\llbracket X \rrbracket$ would be as follows.

$$\llbracket X \rrbracket = \mathcal{L} \mathbf{QR}(X) \cup \mathcal{L} \mathbf{DR}(X) \quad (7.19)$$

The relational trace set is preferable as an extensional description of a DI process because it facilitates the relations defined above, but this choice is only a matter of convenience. Any of the three sets in Equation 7.19 uniquely determines the other two and hence describes the process completely (with one minor exception noted below). If only the quiescent traces of a process $X = (I, O, N) \in \mathbb{D}$ were known, then the divergences could be inferred as

$$\mathcal{L} \mathbf{DR}(X) = ((\mu \lambda p. p \parallel I^1 \parallel (I \cup O)^*) \cup (\mu \lambda q. \bigcup_{i \in \mathbb{N}} \{q \mid i\} \mathcal{L} \mathbf{QR}(X))) - \mathcal{L} \mathbf{QR}(X) \quad (7.20)$$

using the notation for concatenation per Equation 7.3, because every divergence consists of a prefix p of a quiescent trace q followed by an input in I , followed by any sequence of signals (in $(I \cup O)^*$) that is not also a quiescent trace. If only the divergences were known, the quiescent trace set could be inferred as

$$\mathcal{L} \mathbf{QR}(X) = (\bigcup (\mu \lambda d. \bigcup_{i \in \mathbb{N}} \{d \mid i\} \mathcal{L} \mathbf{DR}(X))) - \mathcal{L} \mathbf{DR}(X)$$

being the set of all prefixes of divergences d that are not divergences themselves. The relational trace set also determines the divergences,

$$\mathcal{L} \mathbf{DR}(X) = \{s \in \llbracket X \rrbracket \mid \forall t \in (I \cup O)^*. s \parallel t \in \llbracket X \rrbracket\}$$

because the divergences are the only members of $\llbracket X \rrbracket$ that can be found as a prefix to every possible suffix. (However, this test is inconclusive when the output alphabet is empty, as one might expect from a process that is inherently unresponsive.) The way to infer the remaining one of the three sets by any of these relationships follows trivially from Equation 7.19.

Despite being no more informative than one another in any technical sense, the divergent and quiescent trace sets may be more useful to a designer than the relational trace set for understanding a process, especially when displayed in graphical form. (See Section 4.4.4 for motivation.) An easy way of obtaining the automata to recognize the quiescent and divergent trace sets is by these minor modifications to the relational trace recognizer.

$$\begin{aligned} \mathbf{QR}(X) &= (\lambda(Q, \Sigma, \delta, q_0, F). (Q, \Sigma, \delta, q_0, F - \{q \in Q \mid \forall t \in \Sigma. \delta(q, t) = q\})) \mathbf{RR}(X) \\ \mathbf{DR}(X) &= (\lambda(Q, \Sigma, \delta, q_0, F). (Q, \Sigma, \delta, q_0, F \cap \{q \in Q \mid \forall t \in \Sigma. \delta(q, t) = q\})) \mathbf{RR}(X) \end{aligned}$$

In these expressions, the condition on a state q that $\delta(q, t)$ is equal to q for all $t \in \Sigma$ means that q is a trap state. The relational trace recognizer is always built with a single accepting trap state whose sole purpose is to attract the divergent traces. Excluding trap states from the set of accepting states excludes divergent traces from $\mathcal{L} \mathbf{QR}(X)$ but allows quiescent traces. Making trap states the only accepting states excludes quiescent traces but retains divergences in $\mathcal{L} \mathbf{DR}(X)$.

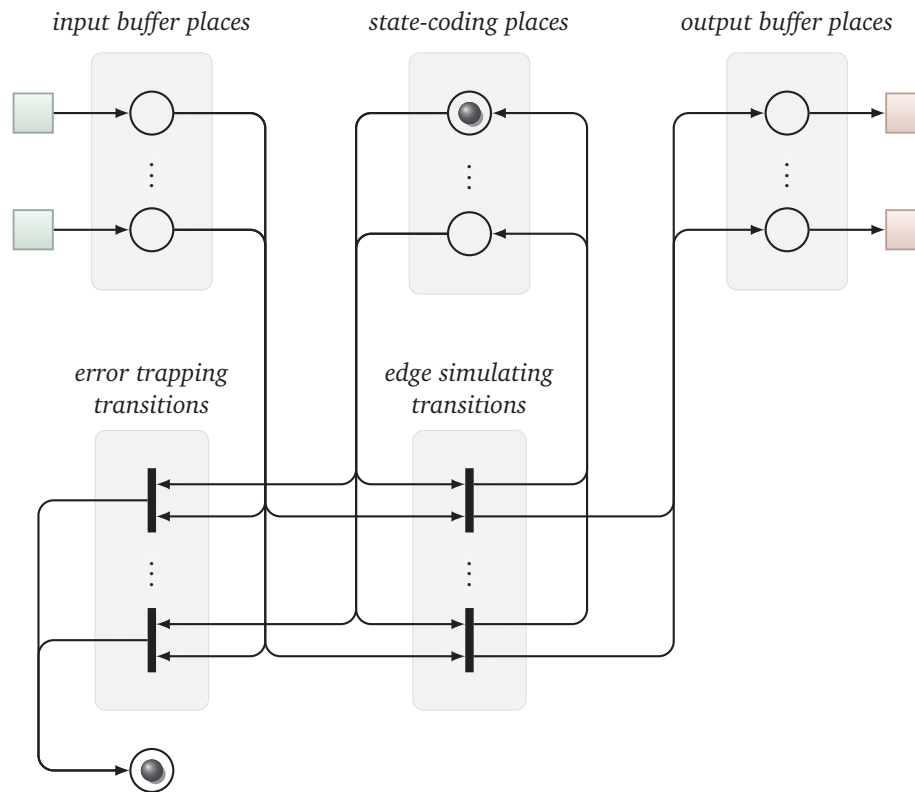


Figure 7.10: simulation of a transducer by an open Petri net with one transition for each edge, a set of places for each state, and arcs organized to implement succession, input, output, and divergence

7.5 A canonical form for Petri nets

As noted at the beginning of this chapter, a transformation from a transducer to an open Petri net model is the last remaining prerequisite for the algebraic closure needed to complete the development of the process combinators started in [Chapter 5](#). In an open Petri net, all input transitions have empty presets and all output transitions have empty postsets, making it a suitable operand for any process combinator. A transformation from transducers to open Petri nets is the subject of this section. Something like a generic template for the transformation is illustrated by an example in [Section 7.5.1](#) and a formal specification follows in [Section 7.5.2](#) and [Section 7.5.3](#).

7.5.1 Overview

Unlike the finite automata discussed in [Section 7.4](#), Petri nets are well suited to describing concurrency, so this construction can start from the transducer model defined originally by [Equation 7.12](#) with concurrent i/o bursts rather than the serial transducer defined by [Equation 7.16](#). The target in general is a Petri net following the basic floorplan shown in [Figure 7.10](#).

- There is an input transition and an adjacent input buffer place for each member of the input alphabet, and a similar arrangement for each output symbol.
- The instantaneous state of the transducer is recorded in the pattern of tokens marking a designated set of state-coding places.
- A change from one state to another is effected by the firing of an edge simulating transition, of which there is one for each edge in the transducer's state graph.
- Each of these transitions is enabled by the buffer places of the input burst labeling the edge it simulates, and by the state coding places representing the state from which the edge originates.
- The postset of each edge simulating transition contains the buffer places for the members of the output burst labeling the edge it represents, so as to transmit the required outputs in due course, and the places encoding the state where the edge terminates.
- Any combination of input signals not covered by the outgoing edges from the current state of the transducer being simulated enables one of the error trapping transitions, whose firing is unsafe by design. In this way, the Petri net models both what the transducer can do and what it can not.

To illustrate the way a Petri net model can be derived from a transducer, a specific example is shown in [Figure 7.11](#). The transducer has three states and two edges. The edge from the initial state 1 to its successor state 2 is labeled by the input burst $\{a, b\}$ and the output burst $\{c, d\}$. The edge from state 2 to the remaining state 0 is labeled by the input burst $\{b\}$ and the output burst $\{e\}$. The final state has no successors. The features of this transducer determine the places, transitions, and arcs in the Petri net as follows.

Places Each of the three states is encoded by a separate place. A buffer place is needed for each of the two inputs a and b , and the three outputs c , d , and e , which is adjacent to the corresponding open transition. An additional marked place forms the common postset of the error trapping transitions, making it unsafe for them ever to fire.

Transitions Each of the two edges in the transducer maps to an anonymous transition. Three more anonymous transitions correspond to the prohibited combinations of states and inputs. These three error trapping transitions simulate divergence in case of prohibited inputs.

- Because state 2 has no outgoing edge labeled by an input burst containing b , the place coding that state and the place adjacent to the input b enable an error trapping transition.
- Because state 0 has no outgoing edges labeled by input bursts containing either a or b , there is an error trapping transition for state 0 with a and for state 0 with b .

If any of these inputs occurs while the Petri net is simulating the relevant state, one of the error trapping transitions is enabled and the Petri net becomes unsafe.

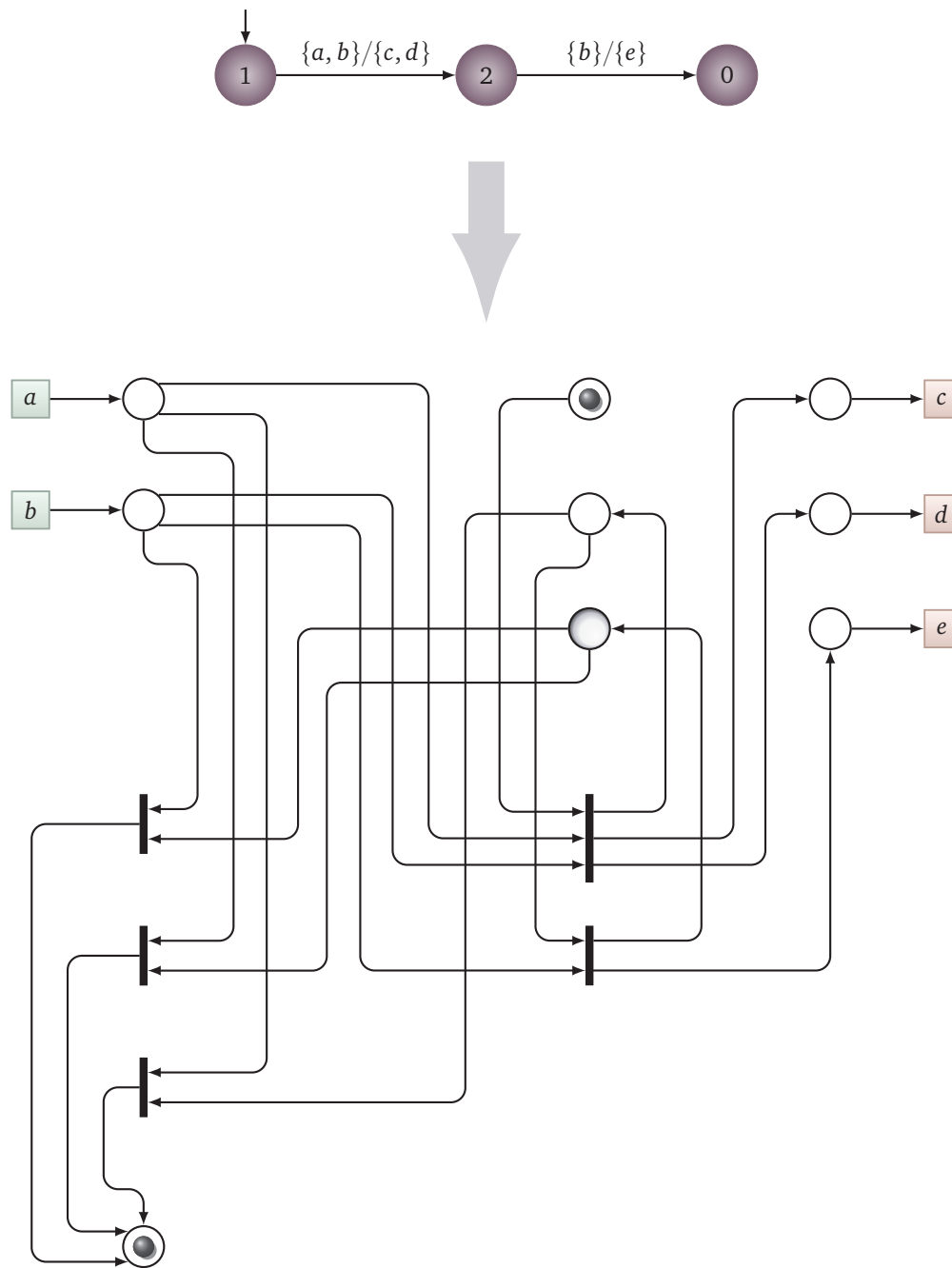


Figure 7.11: A transducer with three states and two edges maps to a Petri net with three state-coding places, two edge simulating transitions, and three error trapping transitions.

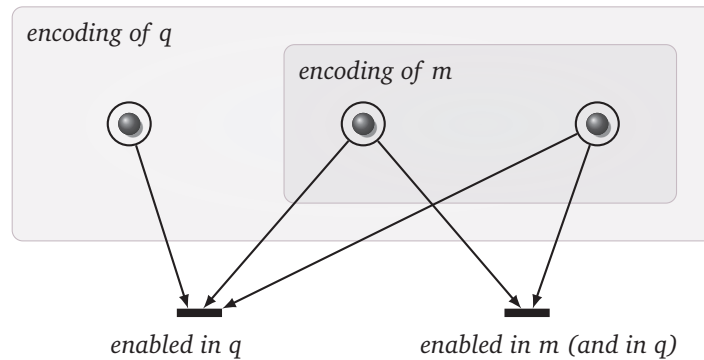


Figure 7.12: If the set of places marked to encode a state q contains the set of places encoding m , then any transitions meant to be enabled during state m are also wrongly enabled during q .

Arcs The arrangement of the remaining arcs in the Petri net provides for it to simulate state changes in the transducer.

- Arcs from the buffer places adjacent to the transitions for a and b are connected to the transition corresponding to the edge labeled by the input burst $\{a, b\}$. An arc from the place encoding state 1 also points to this transition. The transition's postset is made by arcs connected to the buffer places adjacent to the transitions for c and d , and to the place coding state 2.
- The edge simulating transition labeled by the input burst $\{b\}$ has an arc from the place adjacent to the transition for b and an arc from state 2 connected to it. It also has an arc connecting from it to the place adjacent to the transition e because of the output burst $\{e\}$, and an arc to the place coding state 0.

State coding

This example encodes three transducer states with three Petri net places by having exactly one place marked to simulate each state, but for larger state spaces there are other possibilities. With four state-coding places, there are six ways to mark two of them at the same time, and hence the ability to encode up to six states. For five places, the maximum increases to ten states, and for six places, twenty. The numbers grow rapidly, with no more than 23 places needed for 10^6 states, but not as rapidly as the 2^n total possible ways to mark n places. The difference results from a constraint on state encodings whereby no marking may be a proper subset of another.

Without this constraint, a transition enabled in a state m would also be enabled in any state q whose encoding were a superset of that of m , regardless of whether it should be, just because of the way the states are encoded. This situation is illustrated in Figure 7.12. In this sense, state coding is similar to the problem of delay insensitive communication codes (Chapter 13), although the issues of error detection and decoding are probably less relevant to Petri nets than to communication channels.

With so many alternatives, the best way of encoding the states is the next question. The minimum number of state coding places needed to encode n states would be the minimum p satisfying

$$\binom{p}{\lfloor p/2 \rfloor} \geq n \quad (7.21)$$

which is to say the minimum p whose binomial coefficient with the truncated quotient $p/2$ equals or exceeds the number of states n . This number of places is adequate if each state is encoded by a marking in which exactly $\lfloor p/2 \rfloor$ state-coding places are marked. Such an encoding necessarily satisfies the requirement that no marking is a proper subset of any other.

While this choice of encoding would minimize the number of places, it might not minimize the number of arcs, nor the maximum in-degree or out-degree of any vertex, nor their averages, nor the number of tokens in circulation, nor any other complexity or cost metric whose importance in practice is impossible to anticipate. For this reason, the exact specification of a state encoding must be left as an implementation decision.

Complementary inputs

In addition to its trivial state encoding, another misleading aspect of the example in Figure 7.11 is the simplicity of the error trapping transitions. If an input symbol a does not label any outgoing edge from a state m , then a suffices to cause divergence in m , so clearly there is requirement for an error trapping transition enabled by the confluence of m and a , but in general the situation is more complicated. For example, in a circuit implementing a dual-rail decoder [293] with inputs i_0 and i_1 , a state m may allow either i_0 or i_1 but not both, so there would have to be an error trapping transition enabled by the combination of i_0 , i_1 , and m , but not by any proper subset thereof.

More elaborate scenarios are easy to invent. Suppose the input alphabet is $\{a, b, c, d, e\}$, and suppose the following input bursts are acceptable in a state m .

$$\{a, b, c\} \quad \{a, d\} \quad \{a, e\} \quad \{b, c, d\} \quad \{c\} \quad \{d\}$$

Then it follows that there must be no fewer than five error trapping transitions associated with m , where each of them is enabled by one of the following combinations of inputs.

$$\{a, b, d\} \quad \{a, c, d\} \quad \{b, e\} \quad \{c, e\} \quad \{d, e\}$$

For another example, if the following input bursts were acceptable,

$$\{a, b\} \quad \{a, d, e\} \quad \{b, d, e\} \quad \{b, e\} \quad \{c, e\}$$

then these would be the ones to enable the error trapping transitions.

$$\{a, b, d\} \quad \{a, b, e\} \quad \{a, c\} \quad \{b, c\} \quad \{c, d\}$$

The common pattern is that if any of these latter combinations of inputs has been received while the transducer is in state m , then there is no longer any possibility of a valid complement of forthcoming signals to constitute an acceptable input burst for m , because there is bound to be one or another left over. A general solution to this puzzle is needed for coping with arbitrary transducers.



Figure 7.13: Empty transducers map to this Petri net instead of one like Figure 7.10.

Degenerate cases

We should also ask what happens to the template shown in Figure 7.10 when the transducer has no states. Obviously following the same procedure as the example in Figure 7.11 when the transducer is empty would seem to lead to an empty Petri net, or at least to one with no transitions. An empty transducer comes from an empty reachability graph, which is the result of applying the transformations in Chapter 6 to an initially divergent process, but does an empty (or disconnected) Petri net mean the same thing?

A specific interpretation of an empty Petri net is inevitable. A Petri net marking, being only a set of places, is still a marking even if the set is empty. However, it can have no successors if the Petri net is empty because it enables no transitions. The reachability graph of an empty Petri net therefore would consist of a single vertex with no edges, implying a transducer with a single state and no edges. This behavior is more like deadlock than divergence, which is a significant difference even if both are undesirable.



A DI process $X = (I, O, N) \in \mathbb{D}$ can have non-empty alphabets I and O even if the Petri net N is empty, but the conclusion is the same regardless. To follow the conventions mandated in Chapter 5, any process with an empty Petri net N and a non-empty input alphabet I is equivalent to the process $(I, O, N \triangleleft I)$, whose Petri net model $N \triangleleft I$ is the input completion of N by I as defined by Equation 5.19. The Petri net $N \triangleleft I$ would imply only that any possible input in I causes the process to diverge but that it remains quiescent otherwise. (See Section 5.3.4 for further discussion and rationale.) In the reachability graph, there is a tangle of paths consisting of input-labeled and anonymous edges from the initial marking leading ultimately nowhere but to the vertex representing divergence. When these are removed by the divergence propagation transformation described in Section 6.3, the result again is a reachability graph consisting of a single vertex with no edges.

The upshot is that if the transducer is empty, we might as well forget about Figure 7.10 and cut to the chase by generating the Petri net in Figure 7.13 instead. This Petri net does nothing but diverge, just like the one from which the empty transducer must have been derived.

7.5.2 Preparation

The the formal construction of an open Petri net equivalent to a given transducer employs five functions ν_g , ω_g , β_g , σ_g , and ξ_g induced by the transducer $g = \mathbf{T}X$ to generate the edge simulating transitions, error trapping transitions, buffer places, state coding places, and the error place respectively as shown in Figure 7.10, and one further function

$$\bar{i} : \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})) \rightarrow (\mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N}) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{T})))$$

taking the transducer g to a function

$$\bar{i} g : (\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N} \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{T}))$$

such that $(\bar{i} \ g) \ e \in \mathcal{P}(\mathcal{P}(\mathbb{T}))$ covers the set of all prohibited input bursts in any state m of a vertex $(m, e) \in g$ whose adjacency set is e .

We could think of all of these functions except \bar{i} as “vertex factories” in that they generate vertices on demand to populate various parts of the Petri net model of a transducer g . As such, they are constrained to generate mutually distinct vertices. Ensuring this condition raises a similar issue to the serial transducer construction in Section 7.3 with regard to its dependence on an arbitrary fixed injective function $\eta : \mathbb{T} \rightarrow \mathbb{N}$ as a way of avoiding clashes between generated state numbers. A function of this type is needed again in the coming derivation, along with a related device $\mathbb{V}^{\circ^{-1}} : \mathbb{N} \rightarrow \mathbb{V}$, the inverse ordinal function taking any natural number to a distinct vertex in \mathbb{V} per the notation defined by Equation 5.5.

The rest of this section first develops the function \bar{i} needed to specify prohibited input bursts, followed by each of the vertex factories, in preparation for putting them together into a function $\mathbf{P} : \mathbb{D} \rightarrow \tilde{\mathbb{D}}$ expressed in terms of the transducer model in Section 7.5.3. The effect of \mathbf{P} is to take any DI process $X \in \mathbb{D}$ to an equivalent $\mathbf{P}(X) \in \tilde{\mathbb{D}}$ having an open Petri net model.

Prohibited input bursts

Let a process $X = (I, O, N) \in \mathbb{D}$ have an input alphabet I and a transducer $g = \mathbf{T}X$. The transducer contains members of the form (m, e) with each e a set of edges $((i, o), m')$, where $i \subseteq I$ is an input burst. With the acceptable input bursts i given by e and the set to enable the error trapping transitions for m sought, the general requirement is that it cover $\bigcup s$ for a set

$$s = \mathcal{P}\left(\bigcup_{((i,o),m') \in e} \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))\right) - \bigcup \mathcal{P}(i)$$

containing all possible input bursts other than any subset of an acceptable one, where the expression

$$\bigcup \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$$

is the set of input signals mentioned explicitly in the transducer graph g , which may be a subset of I . The set s itself would be a trivial solution to the problem of finding a cover for $\bigcup s$ but is larger than necessary. A minimal sufficient subset with the required coverage is given by

$$s - \bigcup_{b \in s} \{b' \in s \mid b \subset b'\}.$$

The graph g therefore can be said to determine a function $\bar{i} \ g$ that takes any adjacency set e exhibited by g to a set of input bursts $(\bar{i} \ g) \ e$ that suffice to cause undefined behavior.

$$\bar{i} = \lambda g. \lambda e. (\lambda s. s - \bigcup_{b \in s} \{b' \in s \mid b \subset b'\}) (\mathcal{P}(\bigcup_{((i,o),m) \in e} \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))) - \bigcup \mathcal{P}(i)) \quad (7.22)$$

Edge simulating transitions

The Petri net model of a transducer $g = \mathbf{T}(I, O, N)$ requires an anonymous transition to simulate each edge connecting a state m to a state m' labeled by an i/o burst (i, o) associated with g . Hence it requires a distinct member of \mathbb{V} for every possible tuple $(m, ((i, o), m'))$ where $(m, e) \in g$ is a vertex and $((i, o), m')$ is an edge in e . To ensure a distinct member of \mathbb{V} , we can take it to be the

image of a tuple $(m, ((i, o), m'))$ with respect to some injective function from tuples of this type to \mathbb{V} . Any injective function

$$v_g : \mathcal{D}(g) \times ((\mathcal{P}(d) \times \mathcal{P}(r)) \times \mathcal{D}(g)) \rightarrow \mathbb{V}$$

would suffice, where

$$\begin{aligned} d &= \bigcup \{i \in \mathcal{P}(I) \mid (i, o) \in \mathcal{D}(\bigcup \mathcal{R}(g))\} \\ r &= \bigcup \{o \in \mathcal{P}(O) \mid (i, o) \in \mathcal{D}(\bigcup \mathcal{R}(g))\} \end{aligned}$$

are the sets of signals appearing in input and output bursts respectively of the transducer g , but for the sake of concreteness, we can venture an explicit construction in terms of ordinal functions.

Tuples of the form $(m, ((i, o), m'))$ are not totally ordered because they contain members of \mathbb{T} , but an ordering can be inferred for tuples of the form

$$(m, (((\mu \eta) i, (\mu \eta) o), m')) \in \mathcal{D}(g) \times ((\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})) \times \mathcal{D}(g))$$

by fixing an injective function $\eta : \mathbb{T} \rightarrow \mathbb{N}$ as discussed above. These in turn would be members of

$$\mathcal{D}(g) \times (\mathcal{P}((\mu \mu \eta) \mathcal{D}(a)) \times \mathcal{P}((\mu \mu \eta) \mathcal{R}(a))) \times \mathcal{D}(g)$$

where

$$a = \mathcal{D}(\bigcup \mathcal{R}(g))$$

is the set of all i/o bursts appearing in g , suggesting that they could be assigned a unique ordinal

$$(\dot{C}_0 g) (m, (((\mu \eta) i, (\mu \eta) o), m'))$$

in terms of a function \dot{C}_0 defined as

$$\dot{C}_0 = \lambda g. (\lambda a. (\mathcal{D}(g) \times (\mathcal{P}((\mu \mu \eta) \mathcal{D}(a)) \times \mathcal{P}((\mu \mu \eta) \mathcal{R}(a))) \times \mathcal{D}(g))^\circ) \mathcal{D}(\bigcup \mathcal{R}(g))$$

and that a tuple $(m, ((i, o), m'))$ could be uniquely numbered $(C_0 g) (m, ((i, o), m'))$ by a function C_0 defined as follows.

$$C_0 = \lambda g. (\dot{C}_0 g) \circ \lambda(m, ((i, o), m')). (m, (((\mu \eta) i, (\mu \eta) o), m'))$$

Based on the injective function $\mathbb{V}^{\circ-1} : \mathbb{N} \rightarrow \mathbb{V}$ discussed above, the complete specification for v_g follows easily.

$$v_g = \mathbb{V}^{\circ-1} \circ (C_0 g) \tag{7.23}$$

Error trapping transitions

The Petri net simulating the transducer g also needs a distinct error trapping transition for every combination of a state m and a prohibited input burst b where $(m, e) \in g$ is a vertex and $b \in (\bar{i} g) e$ is prohibited in m according to the adjacency set e . Any injective function

$$\omega_g : \mathcal{D}(g) \times \bigcup (\mu \bar{i} g) \mathcal{R}(g) \rightarrow \mathbb{V}$$

would suffice to ensure the uniqueness of the error trapping transition $\omega_g(m, b) \in \mathbb{V}$ for a pair (m, b) relative to other error trapping transitions, but error trapping transitions are also required to be outside the range of edge simulating transitions

$$(\mu \nu_g) \bigcup_{(m,e) \in g} \{m\} \times e$$

by Equation 7.23. An explicit construction of ω_g satisfying both requirements using ordinal functions as above would transform a pair $(m, b) \in \mathcal{D}(g) \times \mathcal{P}(\mathbb{T})$ to a pair $u = (m, (\mu \eta) b) \in \mathcal{D}(g) \times \mathcal{P}(\mathbb{N})$, whose ordinal with respect to the set

$$\mathcal{D}(g) \times (\mu \mu \eta) \bigcup (\mu \bar{i} g) \mathcal{R}(g)$$

is well defined. This value is then offset by the maximum edge simulating transition ordinal

$$\max(\{0\} \cup (\mu C_0 g) \bigcup_{(m,e) \in g} \{m\} \times e)$$

to avoid clashing with any of them, for a result of $(\dot{C}_1 g) u \in \mathbb{N}$ with \dot{C}_1 given by

$$\dot{C}_1 = \lambda g. \lambda u. 1 + (\max(\{0\} \cup (\mu C_0 g) \bigcup_{(m,e) \in g} \{m\} \times e)) + (\mathcal{D}(g) \times (\mu \mu \eta) (\mu \bar{i} g) \mathcal{R}(g))^\circ u$$

or a result $(C_1 g) (m, b) \in \mathbb{N}$ in terms of the original pair $(m, b) \in \mathcal{D}(g) \times \mathcal{P}(\mathbb{T})$ and a function C_1 defined by

$$C_1 = \lambda g. (\dot{C}_1 g) \circ \lambda(m, b). (m, (\mu \eta) b)$$

so that the appropriate error trapping transition factory follows as

$$\omega_g = \mathbb{V}^{\circ-1} \circ (C_1 g). \quad (7.24)$$

Buffer places

Having covered edge simulating transitions and error trapping transitions, we move on to the buffer places needed by the Petri net to simulate the transducer g . Each observable input or output $t \in i \cup o$ in any i/o burst (i, o) labeling an edge $((i, o), m') \in e$ in any adjacency set e of a vertex $(m, e) \in g$ corresponds to an open transition in the Petri net as shown in Figure 7.10, and each of them needs a distinct buffer place adjacent to it.

A buffer place factory $\beta_g : \mathbb{T} \rightarrow \mathbb{V}$ that ensures a distinct place $\beta_g t \in \mathbb{V}$ for each transition $t \in \mathbb{T}$ could be constructed with not much effort as something like

$$\mathbb{V}^{\circ-1} \circ \eta$$

using the functions $\eta : \mathbb{T} \rightarrow \mathbb{N}$ already assumed, but this solution is inadequate because it does not prevent the buffer places from clashing with the edge simulating transitions or error trapping transitions. However, offsetting the value of ηt by the maximum of the error trapping transition ordinal according to the construction above

$$(\mu C_1 g) \bigcup_{(m,e) \in g} \{m\} \times (\bar{i} g) e$$

implicitly prevents a clash with edge simulating transitions as well, suggesting a total $(C_2 g) t$ in terms of a function C_2 given by

$$C_2 = \lambda g. \lambda t. 1 + (\eta t) + \max (\{0\} \cup (\mu C_1 g) \bigcup_{(m,e) \in g} \{m\} \times (\bar{i} g) e)$$

so that a correct buffer place factory function is expressible as follows.

$$\beta_g = \mathbb{V}^{\circ-1} \circ (C_2 g) \quad (7.25)$$

The error place

The initially marked place in the postset of the error trapping transitions in [Figure 7.10](#) is needed only if the set of error trapping transitions

$$\bigcup (\mu \bar{i} g) \mathcal{R}(g)$$

implied by the transducer g is non-empty, so a suitable error place factory $\xi_g \in \mathcal{P}(\mathbb{V})$ need only be a set of at most one vertex. Selecting the vertex arbitrarily as one whose ordinal with respect to $\mathbb{V}^{\circ-1}$ exceeds the maximum of the set

$$(\mu C_2 g) \bigcup_{(i,o) \in \mathcal{D}(\bigcup \mathcal{R}(g))} i \cup o$$

prevents it from clashing with any buffer place, error trapping transition, or edge simulating transition determined by the constructions above. Let the error place factory be defined accordingly as

$$\xi_g = (\mu \mathbb{V}^{\circ-1}) C_3 g \quad (7.26)$$

in terms of the following function C_3 .

$$C_3 = \lambda g. (\lambda s. \langle \{1 + \max (\{0\} \cup (\mu C_2 g) \bigcup_{(i,o) \in \mathcal{D}(\bigcup \mathcal{R}(g))} i \cup o)\}, \emptyset \rangle_{\delta_s \emptyset}) \bigcup (\mu \bar{i} g) \mathcal{R}(g)$$

State coding places

As noted previously, the state coding places $\sigma_g m \in \mathcal{P}(\mathbb{V})$ of a state m in the transducer g must be left as an implementation decision subject to the state coding place factory σ_g satisfying

$$\forall m \in \mathcal{D}(g). \forall m' \in \mathcal{D}(g) - \{m\}. \sigma_g m \not\subseteq \sigma_g m'$$

but whatever way the other vertex factory functions are constructed, the latter must belong to the function space

$$\sigma_g : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{V} - \xi_g - ((\mu \beta_g) \bigcup_{(i,o) \in \mathcal{D}(\bigcup \mathcal{R}(g))} i \cup o) - ((\mu \nu_g) \bigcup_{(m,e) \in g} \{m\} \times e) - (\mu \omega_g) \bigcup_{(m,e) \in g} \{m\} \times (\bar{i} g) e)$$

to avoid generating vertices that clash with them. It is also convenient to have σ_g as a total function of \mathbb{N} in the specification to follow.

$$\forall m \in \mathbb{N}. m \notin \mathcal{D}(g) \Leftrightarrow \sigma_g m = \emptyset \quad (7.27)$$

7.5.3 Specification

Having arrived at satisfactory understandings of state coding, complementary inputs, and certain degenerate cases pertaining to the transformation from transducers to open Petri nets in [Section 7.5.1](#) and [Section 7.5.2](#), we are now in a position to attempt its formal specification, which should flow mostly automatically as a consequence of the way everything is defined up to this point.

Vertices

The set of places P in a Petri net (P, T, A, M, F) simulating a transducer g consists of the union of the buffer places, the error place if any, and the state coding places as discussed previously. These can be given by p_g for a function p defined as follows

$$p = \lambda g. ((\mu \beta_g) \bigcup_{(i,o) \in \mathcal{D}(\bigcup \mathcal{R}(g))} i \cup o) \cup \xi_g \cup \bigcup_{(m,e) \in g} \sigma_g m \quad (7.28)$$

based on [Equation 7.25](#) and [Equation 7.26](#).

Similarly, the set of transitions T is the union of the sets of edge simulating transitions, error trapping transitions, and all signals appearing in any i/o burst. This set is expressible as t_g in terms of a function

$$t = \lambda g. ((\mu \nu_g) \bigcup_{(m,e) \in g} \{m\} \times e) \cup ((\mu \omega_g) \bigcup_{(m,e) \in g} \{m\} \times (\bar{i} g) e) \cup \bigcup_{(i,o) \in \mathcal{D}(\bigcup \mathcal{R}(g))} i \cup o \quad (7.29)$$

based on [Equation 7.22](#), [Equation 7.23](#), and [Equation 7.24](#).

Arcs

Aside from the vertices, there are the arcs A in the Petri net (P, T, A, M, F) . There are enough arcs to be worth partitioning them into five subsets, each expressed as $\bar{a}_i g$ in terms of a function \bar{a}_i for i ranging from 0 to 4.

First are the arcs connecting the state coding places with the edge simulating transitions. We can dispense with these all at once as $\bar{a}_0 g$ where \bar{a}_0 is defined as follows.

$$\bar{a}_0 = \lambda g. \bigcup_{(m,e) \in g} ((\sigma_g m) \times (\mu \nu_g) (\{m\} \times e)) \cup \bigcup_{(b,m') \in e} \{\nu_g(m, (b, m'))\} \times \sigma_g m' \quad (7.30)$$

That is, there is an arc from each member of $\sigma_g m$ to the transition simulating each edge $(m, (b, m'))$ in e , and an arc from the that transition to each member of $\sigma_g m'$. In this way, the firing of any of these transitions unmarks the places encoding m and marks the places encoding m' .

Then there are the arcs in $\bar{a}_1 g$ connecting the state coding and input buffer places to the error trapping transitions.

$$\bar{a}_1 = \lambda g. \bigcup_{(m,e) \in g} (\mu \lambda b. ((\sigma_g m) \cup (\mu \beta_g) b) \times \{\omega_g(m, b)\}) (\bar{i} g) e \quad (7.31)$$

Each prohibited input burst b in $(\bar{i} g) e$ for a state m calls for an arc from every state coding place in $\sigma_g m$ and from the buffer place of every member of b to the error trapping transition $\omega_g(m, b)$. This arrangement enables an error trapping transition precisely when a prohibited combination of inputs from the environment happens concurrently with a state in which they are prohibited.

The arcs from the error trapping transitions to the error place are in $\vec{a}_2 g$, with \vec{a}_2 given by

$$\vec{a}_2 = \lambda g. \bigcup_{(m,e) \in g} ((\mu \omega_g) (\{m\} \times (\bar{i} g) e)) \times \xi_g. \quad (7.32)$$

This one is easy because every error trapping transition due to every combination of states m and prohibited input bursts in $(\bar{i} g) e$ connects only to the error place in ξ_g .

The only arcs left are those connecting the buffer places with the observable transitions and the edge simulating transitions. Taking the input side first, we have a set of arcs $\vec{a}_3 g$ given by

$$\vec{a}_3 = \lambda g. \bigcup_{s \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))} \{(s, \beta_g s)\} \cup \bigcup_{(m,e) \in g} \{\beta_g s\} \times (\mu \nu_g) (\{m\} \times \{((i, o), m') \in e \mid s \in i\}). \quad (7.33)$$

That is, a single arc connects every input transition s to its buffer place $\beta_g s$, and the buffer place $\beta_g s$ is connected by an arc to every transition simulating an edge $((i, o), m')$ whose input burst i contains the signal s . The former marks the buffer place whenever the input transition fires, and the latter inhibits the edge simulating transition when the input s is unavailable. On the output side, we have

$$\vec{a}_4 = \lambda g. \bigcup_{s \in \mathcal{R}(\mathcal{D}(\bigcup \mathcal{R}(g)))} \{(\beta_g s, s)\} \cup \bigcup_{(m,e) \in g} ((\mu \nu_g) (\{m\} \times \{((i, o), m') \in e \mid s \in o\})) \times \{\beta_g s\} \quad (7.34)$$

complementing the inputs, which is mostly the same except that s is an output and the arcs are oppositely directed. Output buffer places do not inhibit the edge simulating transitions but become marked whenever they fire, thus enabling observable output transitions.

The rest

Something still needs to be said about the initial and final markings M and F in the Petri net (P, T, A, M, F) simulating the transducer g . Because the initial state of the transducer is numbered 1 by convention, the initial marking M of the Petri net is $\xi_g \cup \sigma_g 1$, which includes the encoding of state 1 and the error place if any. The final marking is easier, being unconditionally $\sigma_g 0$ because the state corresponding to the final marking is always numbered 0 if there is one. Otherwise, the final marking is \emptyset by Equation 7.27.

Some provision is also needed for the degenerate case of an empty transducer to map to a Petri net like the one shown in Figure 7.13. A Petri net denoted \perp can be constructed for this purpose using the three arbitrarily chosen vertices $v_i = \mathbb{V}^{o-1} i$ for i ranging from 0 to 2 as follows.

$$\perp = (\{v_0, v_1\}, \{v_2\}, \{(v_0, v_2), (v_2, v_1)\}, \{v_0, v_1\}, \emptyset) \quad (7.35)$$

With that, we have everything needed to define the transformation $\mathbf{P} : \mathbb{D} \rightarrow \tilde{\mathbb{D}}$ taking any process $X \in \mathbb{D}$ to its open equivalent $\mathbf{P}(X) \in \tilde{\mathbb{D}}$ based on Equation 7.28 through Equation 7.35.

$$\mathbf{P}(X) = (\lambda(I, O, N). (I, O, (\lambda g. \langle (p g, t g, \bigcup_{i=0}^4 \vec{a}_i g, \xi_g \cup \sigma_g 1, \sigma_g 0), \perp \rangle_{\delta_g^\emptyset} \mathbf{T} X)) X \quad (7.36)$$

7.6 Process combinators revisited

Any of the binary process combinators specified in [Chapter 5](#) under the assumption of open Petri net-modeled processes can be generalized by transforming each operand X and Y to the equivalent canonical forms $\mathbf{P}(X)$ and $\mathbf{P}(Y)$ before composing them. For example, a general form of parallel composition defined as

$$\mathbf{par}(X, Y) = \widetilde{\mathbf{par}}(\mathbf{P}(X), \mathbf{P}(Y))$$

would always work whether X and Y are open or closed. However, this transformation is potentially costly because it requires enumeration of a state space whose size can increase exponentially with the number of Petri net places in X and Y . It is frequently possible to mitigate this cost by computing canonical forms only for members of $\mathbb{D} - \widetilde{\mathbb{D}}$ and leaving members of $\widetilde{\mathbb{D}}$ as they are. A less costly alternative to \mathbf{P} that avoids computing canonical forms unnecessarily

$$\widehat{\mathbf{P}} = \lambda X. (\lambda i. \langle X, \mathbf{P} X \rangle_i) \delta_{\mathbb{D}}^{\widetilde{\mathbb{D}} - \{X\}}$$

allows for the following definitions of the four binary process combinators.

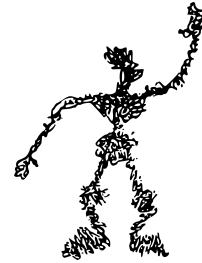
$$\mathbf{seq}(X, Y) = \widetilde{\mathbf{seq}}(\widehat{\mathbf{P}}(X), \widehat{\mathbf{P}}(Y))$$

$$\mathbf{par}(X, Y) = \widetilde{\mathbf{par}}(\widehat{\mathbf{P}}(X), \widehat{\mathbf{P}}(Y))$$

$$\mathbf{alt}(X, Y) = \widetilde{\mathbf{alt}}(\widehat{\mathbf{P}}(X), \widehat{\mathbf{P}}(Y))$$

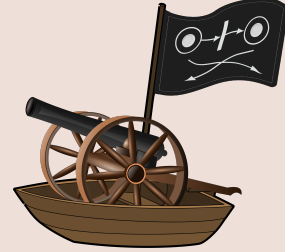
$$\mathbf{env}(X, Y) = \widetilde{\mathbf{env}}(\widehat{\mathbf{P}}(X), \widehat{\mathbf{P}}(Y))$$

These definitions provide the desired algebraic closure of these combinators over \mathbb{D} , allowing us to bid a fond farewell to the wavy lines when putting the combinators to use henceforth.



Canonical curiosia

1. Can the canonical form $\mathbf{P}(X)$ of a process model $X \in \mathbb{D}$ always be inferred from its relational trace set $\llbracket X \rrbracket$ alone? Why or why not? (hint: When is behavioral equivalence not the whole story?)
2. Modify Equation 7.36 to allow for the possibility that the initial and final markings of the Petri net N are equal to each other and non-empty. Is there any justification for ignoring this possibility?
3. Which process refines the other in each of Figure 7.6 and Figure 7.9?
4. What makes each of these “optimizations” superficially plausible but actually too clever by half (except, of course, for the one that is valid)?



- a) simplifying Equation 7.20 to

$$\mathcal{L} \mathbf{DR}(X) = ((\mathcal{L} \mathbf{QR}(X) \parallel I^1) - \mathcal{L} \mathbf{QR}(X)) \parallel (I \cup O)^*$$

- b) allowing $\sigma_g(m) \subset \sigma_g(q)$ if both states m and q have the same i/o bursts labeling their outgoing edges
 - c) transforming $\mathbf{seq}(X, Y)$ to X whenever $X = (I, O, (P, T, A, M, F))$ has an empty final marking $F = \emptyset$
 - d) letting $\hat{\mathbf{P}}$ transform $X = (I, O, N)$ with $N \in \hat{\mathbf{P}} - \tilde{\mathbf{P}}$ directly to an equivalent (I, O, N') with $N' \in \tilde{\mathbf{P}}$ without going through $\mathbf{P}X$ (cf. item 4, page 120)
5. In an example on page 185, a prohibited input burst $\{a, b, d\}$ is a superset of an acceptable input burst $\{a, b\}$. Could an error trapping transition and an edge simulating transition be enabled simultaneously, and if so, how should this condition be interpreted?
 6. A process $\mathfrak{A} \in \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T}) \times \mathbb{P}$ (for “angelic”) with an input a and an output b has the Petri net model $(\{p\}, \{a, b\}, \{(a, p), (p, a), (p, b), (b, p)\}, \{p\}, \emptyset)$ as shown.



- a) What is special about \mathfrak{A} ? (hint: item 3, page 84)
- b) What process combinator expression yields a behavioral equivalent to \mathfrak{A} ? (hint: This question is a kōan.)
- c) What are $\mathbf{RG}(\mathfrak{A})$, $\mathbf{T}(\mathfrak{A})$, $\mathbf{RR}(\mathfrak{A})$, and $\mathbf{P}(\mathfrak{A})$?
- d) Does $\mathfrak{A} \equiv \mathbf{P}(\mathfrak{A})$ hold, and if not, where did it all go wrong? (hint: page 91)

Chaos is inherent in all compounded things. Strive on with diligence.

the Buddha

CHAPTER



BLOCK BUILDING

The extensional style of description developed in various guises over the preceding chapters has been a prerequisite to an informed discussion of circuit semantics, but at some point the rubber needs to meet the road. Ultimately a circuit is a system of components wired together, and the task remains to account specifically for the components and the ways of connecting them. This chapter makes a start at closing the gap between the behavioral description of a circuit and the structural.

Calling a component primitive is another way of saying we choose not to contemplate its inner workings in further detail, but are interested nevertheless in its observable behavior as manifested by the signals it exchanges with its environment. A Petri net-modeled DI process is ideal for describing any primitive component precisely to this extent subject to a few technical provisions. The MERGE, TOGGLE, and JOIN components encountered informally in Part I are examples of primitives, along with a few others to be introduced and specified rigorously in Part III.



When the primitives are understood, a treatment of their combination into potentially useful circuits follows. Two complementary equivalent representations are relevant. The term *hierarchical block* is used in a technical sense in this chapter for one of these two representations. This term is used interchangeably with the term *block diagram*. Either of these refers to a formal model for the graphical depiction of a circuit as interconnected blocks. It consists of either a single primitive component or a collection of smaller block diagrams and primitives. Block diagrams give a full account of the connections among their constituent blocks implicitly as a programmer-friendly algebraic expression, but they also enable a straightforward schematic capture algorithm supportive of user-friendly interactive graphical circuit design tools.

The complementary representation to a block diagram is called a *flat netlist* or a netlist more

briefly. A netlist consists of an enumeration of the primitive components in a circuit and a description of the interconnection in the simplest way possible via numbered wires. A block diagram can always be converted to a netlist. Netlists of any substantial size are neither human readable nor convenient to transform algorithmically, but they are likely to be useful as an interchange format when the finished design is handed off to standard placement, routing, or other technology mapping tools.

Not only can block diagrams be converted to netlists, but transformations among primitive components, block diagrams, netlists, and DI processes are all readily attainable. As a result, it becomes possible to compare one circuit to another or to compare a circuit to a specification by converting both to DI processes and testing for refinement as described in [Chapter 7](#), which implies a refinement relation among circuits themselves. It also becomes possible to make verification more efficient during a bottom-up design by combining each intermediate-level block diagram into a single block before proceeding.

The rest of this chapter is concerned with elaborating on these ideas, but first a few words about lists are appropriate.

8.1 On lists

The nature of this material makes lists more frequently used in this chapter than previously. For example, it is preferable to regard a block in a block diagram as having an ordered sequence of terminals instead of an alphabet. To soften the blow, the concept of a list defined in [Section 7.1](#) is upgraded now with some convenient notation and a few useful operators, while maintaining the model of a list as a function of a natural variable proposed previously. Hence, we may generalize the concepts of the domain and range of a relation ([Equation 5.2](#) and [Equation 5.3](#)) to a list x as

$$\mathcal{D}(x) = \{i \in \mathbb{N} \mid i < |x|\} \quad (8.1)$$

$$\mathcal{R}(x) = (\mu x) \mathcal{D}(x) \quad (8.2)$$

which is to say that the range is defined as the set of its terms using the μ operator defined in [Equation 5.1](#). Note that this usage constitutes a deliberate overloading or abuse of notation.¹ Functional composition of lists, as in $x \circ y$ for lists x and y , may also be invoked without further comment provided that $\mathcal{R}(y)$ is a subset of \mathbb{N} . A list viewed as a function is called injective if it has no duplicate items.

8.1.1 Creating a list

A couple of easy ways to create lists of known simple forms from thin air are helpful enough to have their own notation. The list ι_n (Greek letter iota) is famous for containing every natural number from 0 through $n - 1$ in ascending order.

$$\iota_n = \begin{cases} \epsilon & \text{if } n = 0 \\ \iota_{n-1} \parallel \langle n-1 \rangle & \text{otherwise} \end{cases} \quad (8.3)$$

For example, $\iota_5 = \langle 0, 1, 2, 3, 4 \rangle$. If lists are modeled as functions as discussed above, then ι_n is the identity function for natural numbers less than n but is undefined elsewhere. This device is useful

¹Because functions are not identical to their graphs, $\mathcal{D}(f)$ means something different for a function f than $\mathcal{D}(r)$ does for a relation r . See page [123](#).

already for defining the notation $u^{\underline{n}}$ as the list of n identical copies of u .

$$u^{\underline{n}} = (\lambda i. u) \circ \iota_n \quad (8.4)$$

That is, an underlined natural number as a superscript denotes a list containing that number of copies, as in the example $\rho^{\underline{3}} = \langle \rho, \rho, \rho \rangle$.

It is sometimes helpful to be able to express a list of consecutive numbers whose initial value is greater than zero, for example $\langle 3, 4, 5, 6 \rangle$. Lists of this form could always be written as $(\lambda i. m + i) \circ \iota_n$, where m is the first and $m + n - 1$ is the last number in the list, but they occur frequently enough to be worth defining ι_n^m as a notation for them.

$$\iota_n^m = (\lambda i. m + i) \circ \iota_n \quad (8.5)$$

The example of four consecutive natural numbers in ascending order starting from 3 mentioned above is $\iota_4^3 = \langle 3, 4, 5, 6 \rangle$. This notation is ambiguous insofar as ι_n^m could also mean ι_n composed with itself m times as defined by Equation 6.2, but there is never any reason to compose ι_n with itself. Viewed as a function, ι_n is idempotent, meaning $\iota_n = \iota_n \circ \iota_n = \iota_n \circ \iota_n \circ \iota_n \dots$. Hence there is no downside to the convention of always interpreting ι_n^m according to Equation 8.5.

8.1.2 Deleting from a list

Several operations pertain to the deletion of selected items from a list. For a list x and a natural number n , the expression $x \ll n$, read “ x drop n ” or “ x shifted left n ”, represents the list obtained from x by deleting the first n items.

$$x \ll n = \lambda i. x_{i+n} \quad (8.6)$$

For example $\langle a, b, c, d \rangle \ll 2$ is equal to $\langle c, d \rangle$. It follows from this definition that if n is greater than or equal to the length $|x|$ of a list x , then $x \ll n$ is equal to the empty list ϵ .

On a related note, an expression of the form $x \upharpoonright n$, read “ x take n ”, refers to the list obtained from x by deleting all but the first n items, where x is a list and n is a natural number.

$$x \upharpoonright n = x \circ \iota_n \quad (8.7)$$

For example $\langle a, b, c, d \rangle \upharpoonright 3$ is equal to $\langle a, b, c \rangle$. This definition also depends on the concrete model of lists as functions discussed above.

One further way of deleting items from a list is by way of the **projection** operator used previously in Equation 3.3 on page 48. For a list x and a set s , the expression $x \uparrow s$ denotes the list obtained by deleting all non-members of the set s from x while preserving the relative order of those that are not deleted. The projection operator satisfies the following recurrence.

$$x \uparrow s = \begin{cases} \epsilon & \text{if } x = \epsilon \\ x_0 : (x \ll 1 \uparrow s) & \text{if } x_0 \in s \\ x \ll 1 \uparrow s & \text{otherwise} \end{cases} \quad (8.8)$$

For example, $\langle a, b, c, a, c, b, c, d \rangle \uparrow \{a, c\}$ is equal to $\langle a, c, a, c, c \rangle$.

8.1.3 Folding over a list

Any function $f : (r \times s) \rightarrow s$ taking a pair (u, v) to a result w with $u \in r$ and $v, w \in s$ can be transformed to a function $(\mathcal{F}_c f) : r^* \rightarrow s$ taking a list $x \in r^*$ to a result $y \in s$, where a constant $c \in s$ induces an operator \mathcal{F}_c defined as follows.

$$\mathcal{F}_c(f) = \lambda x. \begin{cases} c & \text{if } x = \epsilon \\ (\lambda(h : t). f(h, (\mathcal{F}_c f) t)) x & \text{otherwise} \end{cases} \quad (8.9)$$

An expression $(\mathcal{F}_c f) \langle x_0, x_1, x_2 \rangle$ is a functional programmer's way of saying $f(x_0, f(x_1, f(x_2, c)))$ without mentioning recursion or iteration. The classic example is $\mathcal{F}_0 \lambda(a, b). a + b$ for a function that sums a list of numbers, although the \mathcal{F} notation used here (Greek letter digamma) is non-standard.

An alternative form of the folding operator without the subscript c is also frequently useful. In this case, the innermost right argument to the operand f is taken as the last item of the list x rather than a constant.

$$\mathcal{F}(f) = \lambda x. (\mathcal{F}_{x_{|x|-1}} f) (x \upharpoonright |x| - 1)$$

For example, $(\mathcal{F} f) \langle x_0, x_1, x_2, x_3 \rangle$ is $f(x_0, f(x_1, f(x_2, x_3)))$. This definition uses the list truncation operator defined in [Equation 8.7](#). A function of the form $\mathcal{F} f$ is defined only for non-empty lists x .

8.1.4 Mapping over a list

A function $f : r \rightarrow s$ induces a function $f^* : r^* \rightarrow s^*$, read “map f ”, that takes a list $x \in r^*$ to a list $y \in s^*$ with $|x| = |y|$, and every y_i equal to $f(x_i)$ for all $0 \leq i < |x|$.

$$f^* = \mathcal{F}_\epsilon \lambda(h, t). (f h) : t \quad (8.10)$$

For example, $f^* \langle a, b, c \rangle$ is equal to $\langle f a, f b, f c \rangle$. The map operator is similar to the μ operator defined by [Equation 5.1](#), except that f^* operates on a lists whereas μf operates on sets.

An astute reader might notice that $f^* x$ is the same as $f \circ x$ if lists are understood as functions, so it might have been simpler to define f^* by itself as $\lambda x. f \circ x$. However, mapping and composition are conventionally regarded as separate concepts, and lists need not be modeled by functions. The definition in [Equation 8.10](#) relies only on the cons operator ([Equation 7.1](#)), and is probably more in keeping with the way this operation might be computed in practice.

8.1.5 Inverse of a list

For an injective list x , the inverse $x^{-1} : \mathcal{R}(x) \rightarrow \mathbb{N}$ is defined as the function that takes an item x_i in $\mathcal{R}(x)$ to the corresponding index $i \in \mathbb{N}$. The inverse function x^{-1} is undefined for arguments outside the range of x . It satisfies the following definition.

$$x^{-1} = \lambda y. |\{i \in \mathcal{D}(x) \mid y \in \mathcal{R}(x \ll i)\}| - 1$$

The inverse of a list x is not a list itself unless x contains only natural numbers less than $|x|$ in addition to being injective. In this case, x is called a **permutation**, and satisfies $x \circ x^{-1} = x^{-1} \circ x = \iota_{|x|}$.

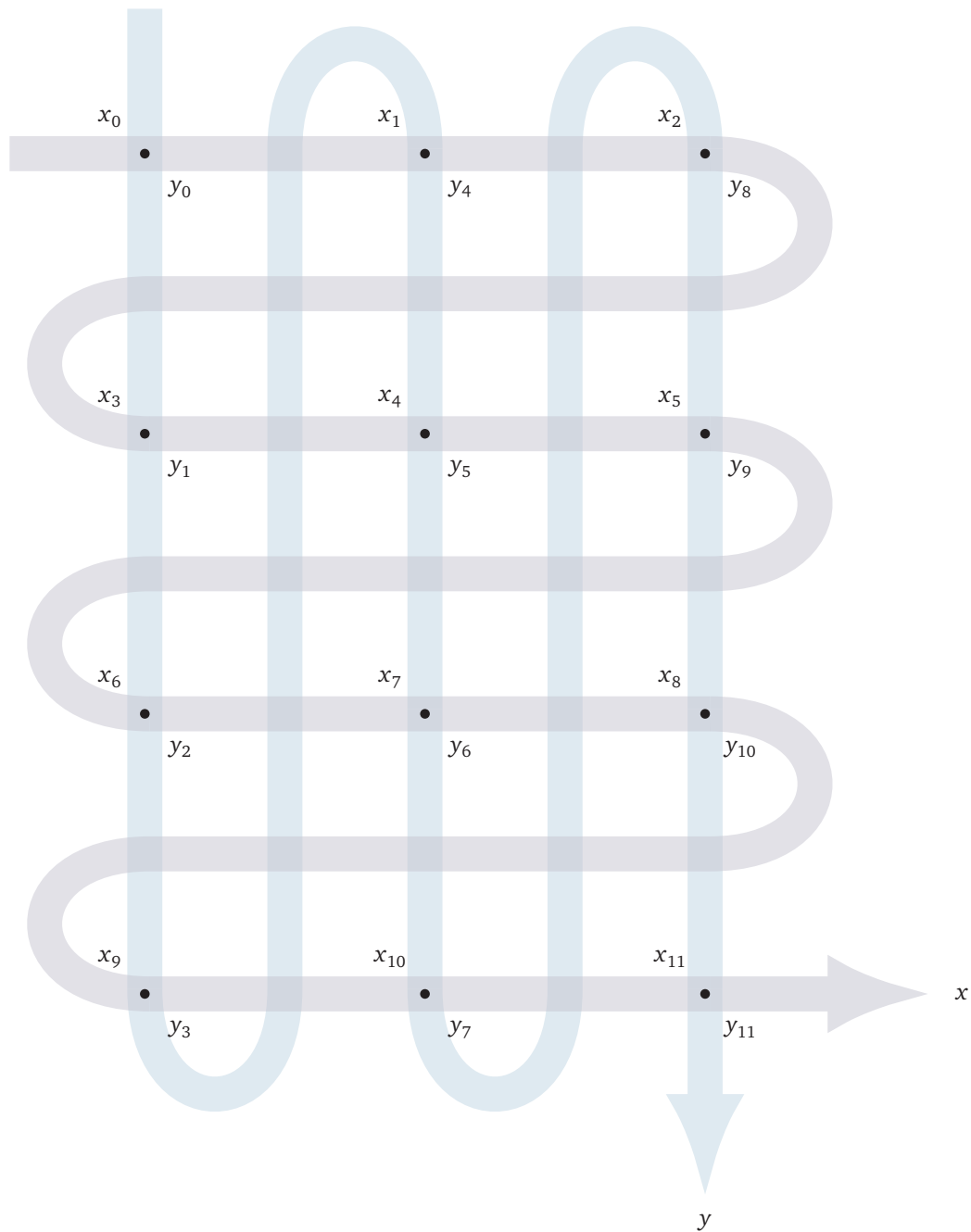


Figure 8.1: A list x of length 12 transposed by 4 is $y = \langle x_0, x_3, x_6, x_9, x_1, x_4, x_7, x_{10}, x_2, x_5, x_8, x_{11} \rangle$, as seen by laying it out in array of 4 rows and traversing it in column-major order.

8.1.6 Flattening a list

A list flattening operator $\flat : S^{**} \rightarrow S^*$ takes any list of lists to its cumulative concatenation.

$$\flat = \mathcal{F}_\epsilon \lambda(h, t). h \parallel t \quad (8.11)$$

This notation is non-standard but readily mnemonic at least to those who are musically inclined. In this example, an empty list in the operand necessarily vanishes.

$$\flat \langle \langle a, b, c \rangle, \langle d, e \rangle, \epsilon, \langle f, g, h, i \rangle, \langle j \rangle \rangle = \langle a, b, c, d, e, f, g, h, i, j \rangle.$$

The list flattening operator may be used to avoid summations in some contexts based on the following identity.

$$|\flat x| = \sum_{i=0}^{|x|-1} |x_i|$$

8.1.7 Transposing a list

One way of reordering the items of a list turns out to be particularly relevant to circuit description, which is to transpose them. Unfortunately there is no standard notation for this operation in the binary form proposed here, so a bit of license is necessary.

If x is a list whose length $|x|$ is divisible by a natural number n , the **transpose** of x by n , denoted $x \times n$, is the list y satisfying $|y| = |x|$ and $y_{i+nj} = x_{j+mi}$ for $i < n$ and $j < m$, where $m = |x|/n$. An expression in closed form for $x \times n$ can be given as follows.

$$x \times n = (\lambda i. (\lambda j. x_j) (|i/n| + (i \bmod n)|x|/n))^* \iota_{|x|} \quad (8.12)$$

The transpose of a list can be visualized as shown in [Figure 8.1](#), by laying out its items in a rectangular array with n rows and m columns in row-major order. The transpose can then be read off by traversing the array in column-major order.

8.2 Primitive blocks

While it is simple enough to envision a block diagram where each primitive block is described by a Petri net, the devil is in the details. Typically multiple distinct instances of the same primitive are deployed in a circuit. An adequately expressive formalism needs to distinguish one instance from another even though they share a common archetype. It would seem natural to model each type of primitive as a process, but it is problematic to speak of two copies of the same process in the style of process models developed up to this point. Process alphabets inhabit a global name space. One process is connected to another when their alphabets intersect. Conversely, keeping two processes separate from each other requires a choice of mutually disjoint alphabets. The closest thing to creating multiple instances of the same process might be to make them mostly the same but to vary the alphabets arbitrarily.

Another small but bothersome problem with modeling a primitive component as a process is that of associating the input and output alphabets of the process with the terminals on the component. A physical device can have a number of terminals, and generally they are not interchangeable:

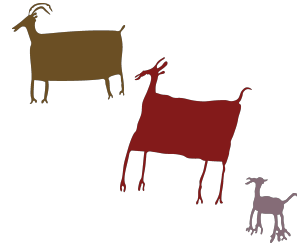




Figure 8.2: an instance of $(1, 1, \lambda(\langle a \rangle, \langle b \rangle). (\{a\}, \{b\}, (\{p\}, \{a, b\}, \{(a, p), (p, b)\}, \emptyset, \emptyset)))$

wiring the first terminal to something and the second to something else can have a different effect from connecting them the other way around. When a process has an input alphabet of $\{a, b, c\}$, no ordering of the alphabet is implied. If b represents the first terminal of a device modeled by this process, a the second, and c the third, then somehow this correspondence needs to be expressed.

The right abstraction to model the general form of a class of primitive components that addresses both of these issues is not exactly a process, but a member of the function space $(\mathbb{T}^* \times \mathbb{T}^*) \rightarrow \mathbb{D}$ meeting certain conditions. A function $B : \mathbb{T}^* \times \mathbb{T}^* \rightarrow \mathbb{D}$ can be constructed to take a pair of lists of alphabet symbols in \mathbb{T} to a process in \mathbb{D} whose input and output alphabets are respectively the ranges of its arguments by Equation 8.2. The correspondence between alphabet symbols and terminals on the associated device is determined by the order of the alphabet symbols in the lists. Multiple distinct instances of the device are expressed by applying B to different arguments, and connected components by applying their respective semantic functions to overlapping arguments.

To consolidate this idea and embellish it slightly, let the universe \mathbb{B} of possible primitive blocks be defined as a set of triples

$$\mathbb{B} = \mathbb{N} \times \mathbb{N} \times ((\mathbb{T}^* \times \mathbb{T}^*) \rightarrow \mathbb{D}) \quad (8.13)$$

where each $(I, O, B) \in \mathbb{B}$ is said to have an **input arity** $I \in \mathbb{N}$, an **output arity** $O \in \mathbb{N}$ and a semantic function B as described above. The additional terms I and O allow us to infer the alphabet cardinalities of any result returned by B without knowing any more about B provided that they satisfy

$$\forall (i, o) \in \mathbb{T}^I \times \mathbb{T}^O. \mathcal{R}(i) \cap \mathcal{R}(o) = \emptyset \Rightarrow B(i, o) \in (\mathcal{R}(i) \times \mathcal{R}(o) \times \mathbb{P}) \cap \mathbb{D} \quad (8.14)$$

where \mathbb{P} is the universe of Petri nets defined by Equation 5.6, and \mathbb{D} is the set of DI processes defined in Section 5.2.5 whose Petri net models are well formed. Hence we stipulate Equation 8.14 as a condition of any well formed primitive block (I, O, B) .

A simple example of a well formed member of \mathbb{B} is the following,

$$\mathbb{I} = (1, 1, \lambda(\langle a \rangle, \langle b \rangle). (\{a\}, \{b\}, (\lambda p. (\{p\}, \{a, b\}, \{(a, p), (p, b)\}, \emptyset, \emptyset))) \text{ min } \forall) \quad (8.15)$$

This block is important subsequently in this chapter, so it is given a name of \mathbb{I} for future reference. It has an input arity of 1 and an output arity of 1. The symbols a and b appearing in Equation 8.15 are bound variables local only to the lambda abstraction defining the semantic function of the block \mathbb{I} , not literally members of the alphabets of any actual instance. In Figure 8.2, the alphabets are $\{q\}$ and $\{r\}$ to emphasize this distinction.

8.3 Hierarchical blocks

Now that primitive blocks are adequately described as members of \mathbb{B} , a systematic way of assembling them into hierarchies is the next priority. A simple solution would be to identify a non-primitive block with a list of primitive blocks as shown in Figure 8.3. An ordering on the terminals of the resulting block is implied by the order of the terminals on the primitives and their positions in the list. This technique extends obviously to hierarchical blocks as nested lists.

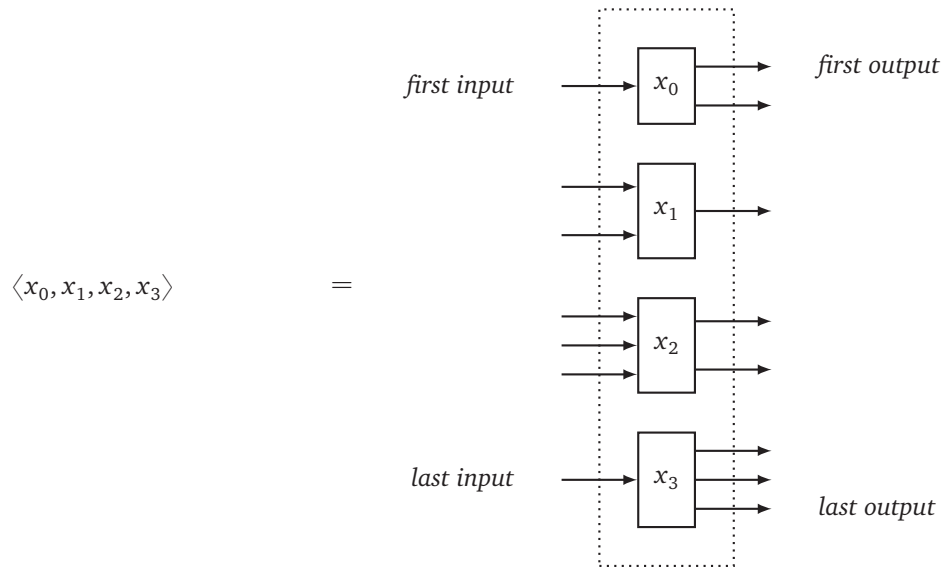


Figure 8.3: A list of primitive blocks $x \in \mathbb{B}^*$ represents a non-primitive block of the members of x with no connections among them and the terminals ordered as shown.

Although simple, this solution suffices completely, barring some possible issues to be noted in [Section 8.3.2](#). The universe of block hierarchies can be defined therefore as the smallest set \mathbb{H} satisfying this recurrence.

$$\mathbb{H} = \mathbb{B} \cup \mathbb{H}^* \tag{8.16}$$

This equation is equivalent to an inductive definition whereby any member of \mathbb{B} is a member of \mathbb{H} , and any list of members of \mathbb{H} is also a member of \mathbb{H} . As usual, a concept of well formed hierarchical blocks is useful. A well formed hierarchical block is one that is non-empty and whose constituent primitive blocks are all well formed according to [Equation 8.14](#). Proceeding on this basis, we need to address the remaining question of how connections between blocks within a hierarchy might be expressed.

Even with the terminals consecutively numbered, it is potentially burdensome in the case of a large network to enumerate the origin and terminus of every connection. A simpler case would be that of networks restricted to one block and one connection. Furthermore, if the connection could be assumed always to be from the first output of the block to its last input as shown in [Figure 8.4](#), then life would be even easier and we could adopt a convention of using the unit list $\langle x \rangle$ to represent the single block x connected in this way. This convention is somewhat at odds with that of letting a list of blocks represent their combination as shown in [Figure 8.3](#), but there is never a need for a combination of just one block, so ambiguity can be avoided simply by reserving unit lists of blocks for the present purpose.



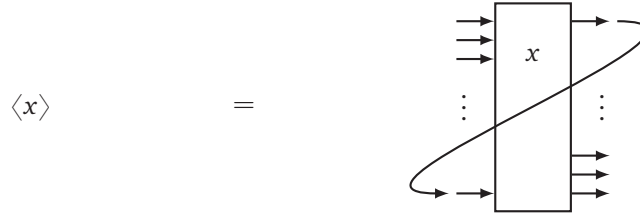


Figure 8.4: A list containing only x represents x with its first output connected to its last input.

8.3.1 Block combinators

With the reader's indulgence, let us temporarily ignore the myriad of other ways one might wish to connect the blocks in a network and formalize what has been proposed so far. It may have some mnemonic value to define $Z : \mathbb{H} \rightarrow \mathbb{H}$ as the first of several **block combinators**.

$$Z = \lambda x. \langle x \rangle \quad (8.17)$$

While Z takes any operand at all and returns the unit list of that operand, it is especially enlightening in an expression of the form $Z(x)$ where x is a block, because then it becomes a pictographic reminder of Figure 8.4 subject to a momentary *gestalt* shift.

It is also fitting to describe the operation of forming a block from a list of blocks as shown in Figure 8.3 by a block combinator, $R : \mathbb{H} \times \mathbb{H} \rightarrow \mathbb{H}$, mnemonic for rack mounting. It is defined as a binary operation, but could be folded over a list by the form $\mathcal{F} R$ as explained in Section 8.1.3. Putting two blocks x and y together can always be done by forming the list of two items $\langle x, y \rangle$, but if x and y are already lists of multiple blocks, it may be advantageous to concatenate them. Concatenation would be conducive to space-efficiency in the concrete representation and to the desirable algebraic property of associativity,

$$R(x, R(y, z)) = R(R(x, y), z)$$

which accords with intuition about an operator that purportedly puts two blocks together without connecting them.

Concatenation of blocks is meaningful only for those given by lists, not for primitive blocks, which are members of \mathbb{B} . It is also incorrect to concatenate unit lists, which represent blocks connected to themselves by the Z combinator as noted above. A definition of R that always does the right thing for well formed (*i.e.*, non-empty) operands would be as follows.

$$R = \lambda(x, y). \begin{cases} x \parallel y & \text{if } x \notin \mathbb{H}^1 \cup \mathbb{B} \wedge y \notin \mathbb{H}^1 \cup \mathbb{B} \\ x \parallel \langle y \rangle & \text{if } x \notin \mathbb{H}^1 \cup \mathbb{B} \wedge y \in \mathbb{H}^1 \cup \mathbb{B} \\ x : y & \text{if } x \in \mathbb{H}^1 \cup \mathbb{B} \wedge y \notin \mathbb{H}^1 \cup \mathbb{B} \\ x : \langle y \rangle & \text{if } x \in \mathbb{H}^1 \cup \mathbb{B} \wedge y \in \mathbb{H}^1 \cup \mathbb{B} \end{cases} \quad (8.18)$$

The condition $x \in \mathbb{H}^1 \cup \mathbb{B}$ means that x is either a unit list or is a member of \mathbb{B} , and therefore is unsuitable for concatenation.

To return to the question of more general forms of interconnection, the answer is closer than might be expected. The combinators R and Z , together with I as defined in Equation 8.15, which may be regarded as a nullary combinator, suffice to express any finite network however complicated. This point is argued more convincingly in Section 8.8. The upshot for the moment is that no loss of generality is incurred by continuing to restrict attention to them.

8.3.2 Block algebra

The two main issues with the concept of block hierarchies as presented hitherto are whether we should be thinking in these terms at all, and if so, whether we have gone about it the right way. The remainder of this section is a brief digression to consider these two questions.

Cleaner models

With regard to the latter question, a cleaner presentation from a mathematical standpoint would postulate three combinators R , Z , and I first as an abstract algebra, and then propose Equation 8.15, Equation 8.16, Equation 8.17, and Equation 8.18 as a *model* for the algebra, acknowledging it to be just one possible model. There is no inherent reason to identify $R(x, y)$ with $\langle x, y \rangle$, or $Z(x)$ with $\langle x \rangle$. It could be argued that these matters should be left as implementation decisions.



The specification of a concrete model for the algebra is not even strictly necessary. For an immaculate presentation, let \mathbb{H} be defined non-constructively as the closure of \mathbb{B} with respect to R and Z , and elide any formal definition of R and Z beyond their type signatures and algebraic laws. In practice this choice would probably amount to block hierarchies being represented as ordered binary trees rather than nested lists.

Mathematically inclined readers opting for an alternative or generalized model are at liberty to do so with minor adjustments to Equation 8.23 and Equation 8.26, which define transformations from \mathbb{H} to \mathbb{L} and \mathbb{B} respectively. Subsequent to this chapter, there is no further reliance on any specific concrete representation for members of \mathbb{H} .

Old school

Regarding the initial question above, a small community of academic researchers influenced by functional programming developed various language-based approaches to the description of network structures in the 1980s and 1990s with emphasis on repetitive arrays [57, 125, 126, 127, 169, 212, 254]. According to these methodologies, the circuit designer was expected to construct derivations and proofs manually using a formal system of algebraic operators so that an implementation could be “extracted” at the conclusion. One of the purest examples can be found in [252]. It would be fair to say that this style was not widely embraced by practicing circuit designers despite much to recommend it [255].

The importance of distinguishing between the roles of the user and the developer might be one of several insights to be gathered in retrospect from this research effort. There is no doubt of the value to a CAD tool developer of being able to describe a network in a form amenable to automated transformation or synthesis at a high level. It does not follow that most circuit designers are mathematicians *manqués* who would rather construct a proof than design a circuit, or that compelling them to do so would lead somehow to better results than their usual practices.

A contemporaneous take on the same problem was that of a *circuit algebra* advocated in [72]. To address the issues of uniquely identified terminals and component instances mentioned at the beginning of Section 8.2, named terminals were an inherent feature of the algebra, and a proposed model for the algebra mandated a universal set of component names as well. A possibly non-injective function specially crafted for each circuit would map its particular component names to a fixed set of basic components. Issues related to naming conflicts prevented the most intuitive choice of

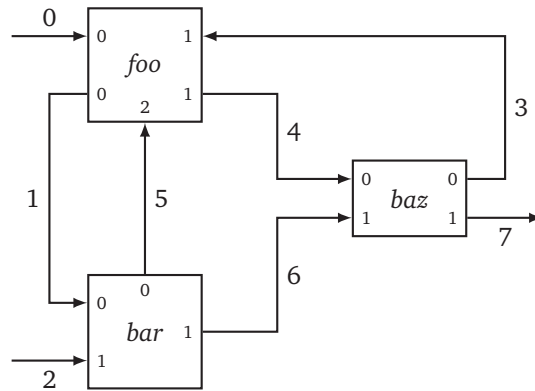


Figure 8.5: a network of globally numbered connections among the primitive blocks *foo*, *bar*, and *baz*, with each block having locally numbered input and output terminals

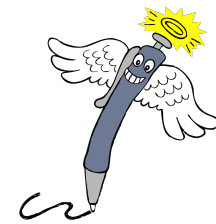
a model from satisfying the algebraic laws, but a model based on structural equivalence classes thereof (*i.e.*, equivalence up to renaming) was shown to be adequate (*cf.* Section B.2.1).

Lest anyone think this topic has run its course, work on algebraic descriptions of network structures and semantics continues apace in far more sophisticated directions involving category theory, with benefits too early to assess at the time of this writing [93, 312].

A perfect description method for network structures would be simultaneously intuitive, rigorous, practical, and loved by all. Its existence is an open question. Fortunately, our present agenda is less ambitious: a serviceable formalism with a clear semantics capable of capturing in writing simple recurrences involving small circuit schematics. To meet even this modest goal, a layer of additional apparatus atop the basic **R**, **Z**, and **I** combinators to be developed starting in Section 8.8 turns out to be unavoidable. These conventions should not be misconstrued as a methodology advocated for circuit designers. At best, they may serve as useful abstractions for CAD tool developers interested in implementing the algorithms explored in Part III.

8.4 Netlists

After primitive blocks and hierarchical blocks, the next important circuit representation is a netlist. A netlist is needed typically as an interchange format for further technology mapping or layout, and is meant to be the closest thing possible to a direct transcription of a circuit schematic. If a schematic were available in graphical form, then it would be laborious but straightforward to construct a netlist from it manually. However, it is more useful to be able to transform a network automatically from a hierarchical block representation to a netlist. Netlists are introduced and formally defined in this section.



8.4.1 Conventions about schematics

A circuit schematic is assumed to incorporate a collection of components, which for our purposes are members of \mathbb{B} , and a set of communication channels connecting them. For ease of discussion, these

channels are called wires, but the theory is independent of the underlying technology. Components in a circuit schematic are depicted as boxes or schematic symbols and wires are depicted as lines or arrows similarly to [Figure 8.5](#).

In a valid circuit schematic, every terminal on every component is connected to exactly one wire. Some wires are connected only to one terminal, and thereby determine the interface between the network and its environment. These wires are called external for this discussion. All other wires are connected from an output terminal on a component to an input terminal on the same or another component. These latter wires are called internal for this discussion.

8.4.2 Specifying a schematic by a netlist

To specify a schematic completely by a netlist, the first step is to assign a unique natural number to each wire. The choice of assignment is completely arbitrary, but the order of the external wires according to the chosen numbering scheme affects the order in which they appear as terminals when the whole network is viewed from the outside as a single block. For example, in [Figure 8.5](#), there are three external wires, consisting of two inputs and one output. One input is on the wire numbered 0, which is connected to the component *foo*, and the other input on the wire numbered 2, which is connected to the component *bar*. If this network were treated as a block, the block would have two input terminals and one output terminal. The first input terminal would be connected internally to *foo*, and the second to *bar*, because 0 precedes 2 in the numbering scheme. A different numbering scheme could have the opposite effect.

It is also necessary for a netlist to distinguish clearly between one terminal and another with respect to each component. It is not enough to say that *foo* is connected to *baz* and *vice versa*. Within each component, the input terminals form an ordered sequence, as do the output terminals. In [Figure 8.5](#), each terminal is labeled explicitly by its ordinal in the sequence. Accordingly, a netlist for this schematic would have to specify that output terminal 1 of *foo* is connected to input terminal 0 of *baz* by wire number 4, and output terminal 0 of *baz* is connected to input terminal 1 of *foo* by wire number 3.

Example of a netlist

To keep all of this information manageable and well organized, a standard form for netlists is helpful. A netlist can be represented as a list of triples (I, O, X) , with one such triple for each component in the circuit. The semantics of the component is given by $X \in \mathbb{B}$, and the numbers of the wires connected to its input and output terminals respectively are stored in lists $I, O \in \mathbb{N}^*$. A netlist to describe the circuit shown in [Figure 8.5](#) could be as follows.

$$\langle (\langle 0, 3, 5 \rangle, \langle 1, 4 \rangle, \text{foo}), \\ (\langle 1, 2 \rangle, \langle 5, 6 \rangle, \text{bar}), \\ (\langle 4, 6 \rangle, \langle 3, 7 \rangle, \text{baz}) \rangle$$

A few points to note about this representation bear reiteration.

- The order of the wire numbers in the lists I and O for each triple is important for identifying each wire number with the right terminal. For example, the input list for *foo* is $\langle 0, 3, 5 \rangle$ and not $\langle 3, 0, 5 \rangle$ or $\langle 5, 3, 0 \rangle$ or any other permutation, because wire 0 is connected to the first input terminal, 3 to the second, and 5 to the third.

- A connection from one component to another can be inferred from the presence of a wire number common to the output list of the former and the input list of the latter. For example, the 5 in the output list of *bar* and the input list of *foo* shows they are connected (by wire 5). Its position in each list indicates the specific terminals of *foo* and *bar* connected by wire 5.
- The order of the triples relative to one another is not important. It would make no difference if the triple containing *foo* were to precede the one containing *bar* in the list. A netlist could just as well be a set, but making it a set would be too unconventional even for a book like this one.

Well formed netlists

A universal set \mathbb{L} of netlists is defined as follows for future reference.

$$\mathbb{L} = (\mathbb{N}^* \times \mathbb{N}^* \times \mathbb{B})^* \quad (8.19)$$

That is, any member of \mathbb{L} is a list of triples (I, O, X) where I and O are lists of input and output wire numbers respectively, and X is a primitive component.

It is worthwhile to standardize certain assumptions about well formed netlists whereby all bets are off if they are not met. For one, a well formed netlist must be non-empty. Furthermore, the component X of every term (I, O, X) in a well formed netlist must be well formed itself by [Equation 8.14](#), and the lengths of the input and output wire lists must match the arities of the component they describe. That is, the condition

$$\forall (I, O, (I', O', B)) \in \mathcal{R}(n). |I| = I' \wedge |O| = O'$$

holds for any well formed $n \in \mathbb{L}$. See the discussion of [Equation 8.13](#) for a reminder about component arities. Finally, there should never be an input connected to an input or an output connected to an output. Hence all input wire lists must be injective and have ranges disjoint from those of other input wire lists. A similar condition applies to the output wire lists. These conditions can be summarized as follows.

$$\begin{aligned} |\mathfrak{b}(\lambda(I, O, X). I)^* n| &= |\mathcal{R}(\mathfrak{b}(\lambda(I, O, X). I)^* n)| \\ |\mathfrak{b}(\lambda(I, O, X). O)^* n| &= |\mathcal{R}(\mathfrak{b}(\lambda(I, O, X). O)^* n)| \end{aligned}$$

That is, the length of a list of all input lists in n laid end to end by the \mathfrak{b} operator defined in [Equation 8.11](#) would equal to the number of distinct terms in it by [Equation 8.2](#), and the same applies to the output lists.

8.5 From hierarchical blocks to netlists

As the preceding exercise suggests, drafting a circuit schematic manually and deriving a netlist from it are tedious work. A block hierarchy $h \in \mathbb{H}$ is potentially a simpler description at a higher level, but is not much use unless it can be transformed to a netlist eventually. To realize its advantages, we undertake in this section to construct a general transformation from block hierarchies to netlists.

This transformation is denoted $\mathcal{T}_{\mathbb{H}\mathbb{L}} : \mathbb{H} \rightarrow \mathbb{L}$ consistently with a naming convention to be used for transformations between any two of \mathbb{H} , \mathbb{B} , \mathbb{D} , and \mathbb{L} . For example, $\mathcal{T}_{\mathbb{B}\mathbb{L}}$ denotes the transformation from \mathbb{B} to \mathbb{L} , and so on. An ensemble of these transformations serves subsequently

in [Section 8.7.3](#) to establish a formal semantics for bisimulation and refinement across the various circuit representations.

Based on the way \mathbb{H} is defined in [Equation 8.16](#), there are only three cases to consider when transforming a member $h \in \mathbb{H}$ to a member $\mathcal{F}_{\mathbb{H}\mathbb{L}}(h) \in \mathbb{L}$. Either it is a primitive block $h \in \mathbb{B}$, a unit list $h \in \mathbb{H}^1$ or a list of some other length. If h is a unit list $\langle h_0 \rangle$, then it is also of the form $\mathbf{Z}(h_0)$, and therefore represents the block h_0 with its first output connected to its last input as shown in [Figure 8.4](#). If it is a list of any other length, then it represents the ordered arrangement of disconnected blocks shown in [Figure 8.3](#). These three cases are now taken in turn.

8.5.1 Primitive blocks

The case of a block $h \in \mathbb{B}$ is the simplest of the three. A primitive block corresponds to a netlist with just one block in it, and one wire connected to each input or output terminal of the block. This result is given directly by $\mathcal{F}_{\mathbb{B}\mathbb{L}}(h)$, where $\mathcal{F}_{\mathbb{B}\mathbb{L}}$ is defined as follows

$$\mathcal{F}_{\mathbb{B}\mathbb{L}} = \lambda(I, O, B). \langle \langle \iota_I, \iota_O^I, (I, O, B) \rangle \rangle \quad (8.20)$$

using the ι operator defined in [Equation 8.3](#). That is, a block can be converted to a netlist containing a single block with external input and output wires numbered consecutively. An example with input arity 2, output arity 3, and a semantic function q is the following.

$$\mathcal{F}_{\mathbb{B}\mathbb{L}}(2, 3, q) = \langle \langle \langle 0, 1 \rangle, \langle 2, 3, 4 \rangle, (2, 3, q) \rangle \rangle$$

8.5.2 Non-unit lists

The next simplest case of the three mentioned above is that of an arbitrary length list $h \in \mathbb{H}^*$ representing a combination of disconnected blocks. The netlist corresponding to the combination is a concatenation of the respective netlists of the individual blocks with the wires renumbered to avoid clashes. This operation is restricted to pairs of netlists at first and then generalized later to lists of them by the \mathcal{F} operator as needed ([Equation 8.9](#)). Denoting this operation by $\mathbf{R}_{\mathbb{L}} : \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{L}$, we can expect it to be something of the form

$$\mathbf{R}_{\mathbb{L}} = \lambda(x, y). x \parallel y'$$

where y' is y with the wires renumbered so that no wire number in y' coincides with any wire number in x .

It remains to specify the renumbering of y . A simple way of renumbering y to prevent clashes with x would be to add a sufficiently large constant offset to every wire number in y . Clearly the successor of the maximum wire number in x

$$1 + \max \bigcup_{(I, O, X) \in \mathcal{R}(x)} \mathcal{R}(I \parallel O)$$

would be sufficient as an offset. A function

$$f = (\lambda i. i + 1 + \max \bigcup_{(I, O, X) \in \mathcal{R}(x)} \mathcal{R}(I \parallel O))^*$$

to add this number to every term in a list would enable each term $(I, O, X) \in \mathcal{R}(y)$ to be rewritten as

$$\lambda(I, O, X). (f I, f O, X)$$

and all of y to be renumbered consistently to

$$(\lambda(I, O, X). (f I, f O, X))^* y$$

without altering any of its connections. This reasoning suggests the following definition for \mathbf{R}_{\perp} .

$$\mathbf{R}_{\perp} = \lambda(x, y). x \parallel ((\lambda f. \lambda(I', O', X). (f I', f O', X)) (\lambda i. i + 1 + \max_{(I, O, X) \in \mathcal{R}(x)} \bigcup \mathcal{R}(I \parallel O)))^* y \quad (8.21)$$

8.5.3 Unit lists

The remaining of the three cases to be considered when transforming a hierarchical block $h \in \mathbb{H}$ to a netlist $\mathcal{F}_{\mathbb{H}\perp}(h) \in \mathbb{L}$ is that of a unit list $h \in \mathbb{H}^1$. As noted previously, a block of the form $\langle h_0 \rangle$ corresponds to $\mathbf{Z}(h_0)$, which is to say h_0 with its first output connected to its last input. Hence if $x \in \mathbb{L}$ is a netlist representation of h_0 , then a netlist representation of $\mathbf{Z}(h_0)$ may differ from x by as little as the change of a single output wire number to a particular input wire number. Let this netlist be given by $\mathbf{Z}_{\perp}(x)$ for a function $\mathbf{Z}_{\perp} : \mathbb{L} \rightarrow \mathbb{L}$ yet to be determined.

First and last terminals have a well defined meaning in reference to a netlist. For any $x \in \mathbb{L}$, the wire ranges $(i, o) = u_0(x) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$ are obtained by a function $u_0 : \mathbb{L} \rightarrow \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$ defined as follows.

$$u_0 = \lambda x. (\lambda s. (\bigcup \mathcal{D}(s), \bigcup \mathcal{R}(s))) \mathcal{R}((\lambda(I, O, X). (\mathcal{R}(I), \mathcal{R}(O))))^* x)$$

Their intersection $i \cap o$ is the set of internal wires, the difference $o - i$ is the set of external output wires, and the difference $i - o$ is the set of external input wires. The first output from x is therefore numbered $t = \min(o - i)$ and the last input, $\max(i - o)$.

A simple way of defining $\mathbf{Z}_{\perp}(x)$ would be as a function that rewrites t to $\max(i - o)$ wherever it occurs in any output wire list O of a term $(I, O, X) \in \mathcal{R}(x)$ while keeping all other wire numbers $w \in \mathcal{R}(O)$ invariant, which would mean rewriting O to

$$(\lambda w. (\lambda k. \langle w, \max(i - o) \rangle_k) \delta_w^t)^* O$$

but then t would be unused as a wire number in the result. A tidier solution would close the gap by decrementing every external output wire number other than t in an output wire list. Any wire number w in $o - i$ would satisfy $i - \{w\} = i$ and therefore could be rewritten to

$$w - \delta_i^{i - \{w\}}$$

when it differs from t by rewriting O to

$$(\lambda w. (\lambda k. \langle w - \delta_i^{i - \{w\}}, \max(i - o) \rangle_k) \delta_w^t)^* O.$$

This reasoning suggests a rewrite rule $u_1 u_0 x$ applicable to every output list O in x , in terms of a function $u_1 : \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \rightarrow (\mathbb{N}^* \rightarrow \mathbb{N}^*)$ given by

$$u_1 = \lambda(i, o). (\lambda t. (\lambda w. (\lambda k. \langle w - \delta_i^{i - \{w\}}, \max(i - o) \rangle_k) \delta_w^t)^*) \min(o - i)$$

and the corresponding definition for \mathbf{Z}_{\perp} .²

$$\mathbf{Z}_{\perp} = \lambda x. (\lambda(I, O, X). (I, (u_1 u_0 x) O, X))^* x \quad (8.22)$$

²erratum: u_1 as originally defined above is incorrect, but defining $u_1 = \lambda(i, o). \lambda w. (\lambda k. \langle w, \max(i - o) \rangle_k) \delta_w^t$ suffices.

8.5.4 The transformation

A recursive definition of $\mathcal{T}_{\mathbb{H}\mathbb{L}}$ now follows directly from Equation 8.20, Equation 8.21, Equation 8.22 and the folding operator defined in Equation 8.9.

$$\mathcal{T}_{\mathbb{H}\mathbb{L}}(h) = \begin{cases} \mathcal{T}_{\mathbb{B}\mathbb{L}} h & \text{if } h \in \mathbb{B} \\ \mathbf{Z}_{\mathbb{L}} \mathcal{T}_{\mathbb{H}\mathbb{L}} h_0 & \text{if } h \in \mathbb{H}^1 \\ (\mathcal{F} \mathbf{R}_{\mathbb{L}}) \mathcal{T}_{\mathbb{H}\mathbb{L}}^* h & \text{otherwise} \end{cases} \quad (8.23)$$

This transformation enables us to convert any hierarchical block $h \in \mathbb{H}$ to an equivalent netlist in \mathbb{L} .

8.6 From hierarchical blocks to primitive blocks

Although transforming a block diagram to a netlist as described in the previous section is necessary when the work on a design is finished, something like the opposite needs to be done most of the time up until then. Progress during the work is made by suppression of distracting or repetitious details and abstraction of common patterns, which is achieved only through a diligent practice of hierarchical organization. As it stands, our theory of block diagrams poses no impediment to treating an arbitrarily complex hierarchical block $h \in \mathbb{H}$ nominally as a primitive in subsequent use, but the theory can do better by temporarily making a hierarchical block's primitive status official, which is to say by transforming it to an equivalent primitive block $\mathcal{T}_{\mathbb{H}\mathbb{B}}(h) \in \mathbb{B}$.

This transformation is advantageous because it allows the whole block to be modeled by a single Petri net instead of a large system of small Petri nets interacting with one another. Often the Petri net model of a whole block can be dramatically simpler after optimization than the sum of the models of its constituent blocks. (See Section 9.2.) This effect is to be expected when the circuit observes a simple protocol with its environment while hiding its inner complexity, as any well designed circuit should. A simpler Petri net model implies more efficient simulation, and is a prerequisite to the method of refinement checking by trace analysis to be discussed in Section 8.7.

The transformation $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ itself is simpler than the foregoing $\mathcal{T}_{\mathbb{H}\mathbb{L}}$ due to having only two cases in need of consideration. If a block $h \in \mathbb{H}$ is already a member of \mathbb{B} , then there is nothing to be done. We are left only with unit lists $h \in \mathbb{H}^1$, which connect the first output of h_0 to its last input, and lists of other lengths, which represent a collection of disconnected blocks (*cf.* Section 8.5).

8.6.1 Non-unit lists

The case of a list $h \in \mathbb{H}^*$ with $|h| \neq 1$ pertains to a circuit containing mutually isolated blocks as shown in Figure 8.3. An equivalent primitive block would be modeled by a Petri net consisting of $|h|$ disconnected components. Because the transformation $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ will have been applied recursively to each of the blocks, the current task reduces to one of combining a list of primitives in \mathbb{B} into a single primitive. It is reduced further by restricting it to a binary operation on \mathbb{B} , which can be generalized later to lists by folding (Equation 8.9). Let a function $\mathbf{R}_{\mathbb{B}} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ denote this operation.

To construct $\mathbf{R}_{\mathbb{B}}$, we need to account for the block arities and the semantic function. For a pair of blocks $(I, O, B), (I', O', B') \in \mathbb{B}$, it should be clear that putting them in parallel with no connections between them would result in a block $\mathbf{R}_{\mathbb{B}}((I, O, B), (I', O', B'))$ with $I + I'$ inputs and $O + O'$ outputs.

Combining the semantic functions B and B' is almost as easy. Recall from Section 8.2 that a block in \mathbb{B} is modeled by a function $B : \mathbb{T}^* \times \mathbb{T}^* \rightarrow \mathbb{D}$ that takes a pair of lists of alphabet symbols to a process in \mathbb{D} having those symbols in its alphabets. We must therefore build a semantic function

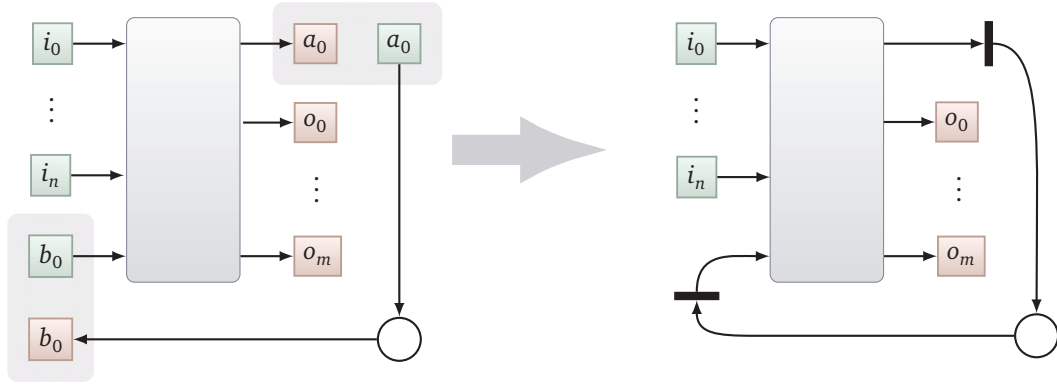


Figure 8.6: Connecting the first output of a Petri net $B(i \parallel \langle b_0 \rangle, \langle a_0 \rangle \parallel o)$ to the last input can be done by parallel composition with a wire whose terminals a_0 and b_0 match those to be connected.

from B and B' that takes pair of lists (i, o) as an argument whose lengths respectively are $I + I'$ and $O + O'$. To keep the terminals in the right order, the first I terms in the list of input alphabet symbols i should be fed to B , and the remaining I' inputs to B' . A similar condition applies to the output alphabets. For useful choices of i and o (meaning disjoint in their ranges and both injective), the alphabets of $B(i \parallel I, o \parallel O)$ and $B'(i \ll I, o \ll O)$ are disjoint by Equation 8.14. Forming their parallel composition using the **par** process combinator (Section 5.4.2) results therefore in a Petri net model with two disconnected components, as required, suggesting the following definition for $\mathbf{R}_{\mathbb{B}}$.

$$\mathbf{R}_{\mathbb{B}} = \lambda((I, O, B), (I', O', B')). (I + I', O + O', \lambda(i, o). \mathbf{par} (B(i \parallel I, o \parallel O), B'(i \ll I, o \ll O))) \quad (8.24)$$

8.6.2 Unit lists

A block $h \in \mathbb{H}^1$ of unit length represents a circuit derived from that of h_0 by connecting the first output to the last input. We can assume that h_0 is a member of \mathbb{B} and express the corresponding result for h as $\mathbf{Z}_{\mathbb{B}}(h_0)$ in terms of function $\mathbf{Z}_{\mathbb{B}} : \mathbb{B} \rightarrow \mathbb{B}$ that edits the block representation accordingly. Because of the connection, $\mathbf{Z}_{\mathbb{B}}(h_0)$ has one less input terminal and one less output available to the environment than h_0 does, so $\mathbf{Z}_{\mathbb{B}}$ must decrement the arities as shown,

$$\mathbf{Z}_{\mathbb{B}} = \lambda(I, O, B). (I - 1, O - 1, u_3 B) \quad (8.25)$$

along with transforming the semantic function B to some modified semantic function $u_3 B$ by a function

$$u_3 : ((\mathbb{T}^* \times \mathbb{T}^*) \rightarrow \mathbb{D}) \rightarrow ((\mathbb{T}^* \times \mathbb{T}^*) \rightarrow \mathbb{D})$$

to be determined presently.

An easy and robust modification to the semantic function would have it generate a small additional Petri net similar to that of Figure 8.2 connected like a wire to the original Petri net generated by B . When the input and output terminals of the wire are chosen to coincide respectively with the first output and the last input of the original, their parallel composition forms the required connection. This operation is depicted in Figure 8.6.

The additional Petri net should be of the form $B'(a, b)$ for unit alphabet lists $a = \langle q \rangle$ and $b = \langle r \rangle$ with the semantic function B' being that of the nullary combinator $\mathbb{I} = (I', O', B')$ by Equation 8.15, and the alphabet symbols $p, q \in \mathbb{T}$ chosen not to clash with any that might be supplied in the argument (i, o) to B , the semantic function in Equation 8.25 we seek to modify. To ensure the latter condition, we need to invoke the function $\gamma : \mathcal{P}(\mathbb{T}) \rightarrow \mathbb{T}$ satisfying $\gamma s \notin s$ as postulated in Section 5.2.3. Then any $(i, o) \in \mathbb{T}^* \times \mathbb{T}^*$ induces a pair $(a, b) = u_2(i, o) \in \mathbb{T}^1 \times \mathbb{T}^1$ of unit lists of symbols different from those in i and o when

$$u_2 : \mathbb{T}^* \times \mathbb{T}^* \rightarrow \mathbb{T}^1 \times \mathbb{T}^1$$

is defined as

$$u_2 = \lambda(i, o). (\lambda s. (\lambda r. (\lambda q. (\langle q \rangle, \langle r \rangle)) \gamma (s \cup \{r\}))) \gamma s) \mathcal{R}(i) \cup \mathcal{R}(o).$$

In keeping with the reduction in arity indicated by Equation 8.25, the modified semantic function $u_3 B$ must take a pair of lists (i, o) whose lengths are one less than the lengths of the lists required by the original semantic function B . Extending a pair (i, o) meant for $u_3 B$ to a pair $(i \parallel b, a \parallel o)$, where (a, b) is $u_2(i, o)$, makes it an appropriate argument for B . Evaluating $B(i \parallel b, a \parallel o)$ results in a Petri net whose first output and last input are respectively the specific values $q = a_0$ and $r = b_0$. Applying the semantic function of the \mathbb{I} combinator defined in Equation 8.15 to the same pair (a, b) would generate a Petri net whose parallel composition with $B(i \parallel b, a \parallel o)$ anonymizes these signals and leaves it in the desired form shown in Figure 8.6. Hence we can define u_3 in full as follows,

$$u_3 = \lambda B. \lambda(i, o). (\lambda(a, b). \mathbf{par} (B(i \parallel b, a \parallel o), ((\lambda(I', O', B'). B') \mathbb{I}) (a, b))) u_2(i, o)$$

thus completing the definition of $\mathbf{Z}_{\mathbb{B}}$ in Equation 8.25.

8.6.3 The transformation

In summary, a primitive block $\mathcal{T}_{\mathbb{H}\mathbb{B}}(h) \in \mathbb{B}$ equivalent to an arbitrary hierarchical block $h \in \mathbb{H}$ can be obtained by Equation 8.24, Equation 8.25 and the following recurrence.

$$\mathcal{T}_{\mathbb{H}\mathbb{B}}(h) = \begin{cases} h & \text{if } h \in \mathbb{B} \\ \mathbf{Z}_{\mathbb{B}} \mathcal{T}_{\mathbb{H}\mathbb{B}} h_0 & \text{if } h \in \mathbb{H}^1 \\ (\mathcal{F} \mathbf{R}_{\mathbb{B}}) \mathcal{T}_{\mathbb{H}\mathbb{B}}^* h & \text{otherwise} \end{cases} \quad (8.26)$$

8.7 From blocks and netlists to processes

The effort in this chapter to develop the transformations $\mathcal{T}_{\mathbb{B}\mathbb{L}}$, $\mathcal{T}_{\mathbb{H}\mathbb{L}}$ and $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ combined with the title of this section may seem like the beginning of a campaign to develop transformations in both directions between every possible pair of circuit representations including \mathbb{D} , but we need not go that far. Netlists are a low level target representation with no need to be converted back to block diagrams, and conversion from \mathbb{B} to \mathbb{H} is trivial because \mathbb{B} is a subset of \mathbb{H} . Conversion from \mathbb{D} to \mathbb{H} in general deserves to be called circuit synthesis, which is certainly useful but well beyond the scope of this chapter and deferred to Part IV. A conversion from \mathbb{D} to \mathbb{B} shows up in Chapter 9 for reasons motivated therein, but otherwise only the conversions from \mathbb{B} , \mathbb{H} , or \mathbb{L} to \mathbb{D} remain, and only if there is some good reason to care about them.

Those of a cautious disposition might find reason enough as follows. If a block diagram or netlist is designed manually, there is always a possibility of human error, and if it is generated

algorithmically, there is always a possibility of the algorithm being incorrect. If the algorithm has been proved correct, there is always a possibility of a bug in the implementation [144], and if no bugs affect a particular production run, there is always that possibility in the next one. It can be difficult or costly to discern by inspection whether any of these possibilities has been realized.

To mitigate some of this uncertainty, it would be helpful to have the tools to test for behavioral equivalence and refinement between circuits in other representations, just as we already can for DI processes as members of \mathbb{D} according to Equation 7.18. If behavioral equivalence could be established between a simple trustworthy version of a netlist or block diagram and a painstakingly hand optimized version, some confidence in the latter would be justified. If a new method of generating families of block diagrams algorithmically or according to some template is under development, it is useful to the developer be able to judge the test results more effectively than by manual simulation.

Having opened this can of worms, one can not help but notice that any verification algorithm is subject in principle to some of the same weaknesses as the design methodology it purports to verify. However, with due care and attention, it can at least curtail the class of undetected errors to those resulting from a malicious collaboration between separate bugs in independent circuit synthesis and verification algorithms. Such an error would require a synthesis algorithm to generate a plausible but defective design, and a generally reliable verifier to disregard defects of that very same nature.³

Hence the work of defining these transformations is motivated by a strategy for verification based on extending the concept of refinement to any pair of circuits or processes $X, Y \in \mathbb{L} \cup \mathbb{H} \cup \mathbb{D}$ by transforming them to members of \mathbb{D} as needed and then leveraging the definition of refinement given by Equation 7.18. To this end, the first thing we should ask is whether there is anything half baked about the idea of converting a circuit to a process. Never mind that multiple Petri net models N could be behaviorally equivalent to one another; no circuit uniquely determines a process $(I, O, N) \in \mathbb{D}$ because the alphabets I and O still have to be formally modeled somehow. A casual answer is to choose them arbitrarily, but then nothing prevents equivalent circuits from being converted to incompatible processes due to different choices of alphabets, thereby defeating the purpose of this endeavor.

Doing something about the alphabets is the subject of Section 8.7.1, using them in transformations $\mathcal{T}_{\mathbb{B}\mathbb{D}}$, $\mathcal{T}_{\mathbb{H}\mathbb{D}}$, and $\mathcal{T}_{\mathbb{L}\mathbb{D}}$ is done in Section 8.7.2, and with that out of the way, a generalized refinement relation is formulated in Section 8.7.3.

8.7.1 Alphabet soup

Any choice of alphabets assigned to the process derived from a circuit in a different representation is arbitrary to some extent. The trick is to choose them consistently enough to avoid introducing gratuitous differences between otherwise equivalent circuits. A first step in this direction is to restrict all of the chosen alphabet symbols to a set $\mathbb{G} \subset \mathbb{T}$, mnemonic for “generic”, where \mathbb{T} is the universe of alphabet symbols used by processes in \mathbb{D} as explained in Section 5.2.1. There is no telling in advance how many generic alphabet symbols might be required so we need \mathbb{G} to be countably infinite, which is achievable by defining it as



$$\mathbb{G} = \bigcup_{n \in \mathbb{N}} (F_{\emptyset} \lambda(h, t). t \cup \{\gamma t\}) \iota_n \quad (8.27)$$

³See [12] for the seminal reference in support of software reliability through the use of multiple independent algorithms, [141, 142] for lively polemics on the subject, and [10] for a more modern perspective.

using the γ function discussed in Section 8.6.2. We can also define a total ordering on \mathbb{G} such that a generic symbol $a \in \mathbb{G}$ precedes $b \in \mathbb{G}$ precisely when this condition holds

$$\exists n \in \mathbb{N}. a \in g \ n \wedge b \notin g \ n \quad (8.28)$$

where $g : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{G})$ is defined as

$$g = \lambda n. (f_{\emptyset} \lambda(h, t). t \cup \{\gamma t\}) \iota_n$$

so that it is meaningful to refer to the ordinal of a generic symbol a as $\mathbb{G}^\circ a$, and to the n -th generic symbol as $\mathbb{G}^{\circ-1} n$ using the ordinal notation defined in Section 5.1.4. If a block or a netlist is transformed by convention to a process with a generic alphabet such that the n -th terminal on the circuit corresponds to the n -th generic symbol, then the processes derived from any two circuits should be equivalent whenever the circuits are equivalent.

Alphabet orderings

In addition to comparing two circuits, we might also want to compare a circuit to a known process, perhaps by converting the circuit to a process first and then comparing the two processes, but this approach is problematic insofar as a process with a generic alphabet derived from a circuit might not match the alphabet of the process with which it is to be compared. Associating a different arbitrarily chosen alphabet with the circuit could imply a different conclusion about their equivalence. This problem is partly solved by transforming the alphabet of the given process to a generic form so that it coincides with that of the process derived from the circuit, provided the arities are compatible, but not completely solved because there is generally more than one way to rewrite the alphabet of a process. For example, a process with an input alphabet $\{a, b\}$ could be transformed to one where a is replaced by $\mathbb{G}^{\circ-1} 0$ and b is replaced by $\mathbb{G}^{\circ-1} 1$, or the other way around. Not every circuit with two inputs supports using them interchangeably, so the assignment of the symbols could make a difference as to whether it is equivalent to the process under consideration.

There is no satisfactory solution to the latter part of this problem except to recognize that it is ill posed. Whether a process is equivalent to a circuit rightly depends on which terminal of the circuit is assigned to each member of the process's alphabet. It should be no surprise that a different assignment may imply a different conclusion because neither the terminals nor the alphabet symbols are interchangeable in general with respect to the behavior.

The closest we can come to a sensible question about equivalence between processes and circuits is therefore to ask whether they are equivalent up to a specified **alphabet ordering**. For a process $X = (I, O, N) \in \mathbb{D}$, any injective function $\alpha \in \mathbb{T}^*$ satisfying $I \cup O \subseteq \mathcal{R}(\alpha)$ determines an alphabet ordering on X in that a symbol x precedes a symbol y under the ordering whenever $\alpha^{-1} x$ is less than $\alpha^{-1} y$. It is easy to construct an alphabet ordering for a known process just by making a list of its alphabet symbols wherein each occurs exactly once, and just as easy to construct a different alphabet ordering for the same process by listing them in a different order. Listing them in the order of the terminals associated with them on a circuit of interest establishes a context for settling questions about equivalence.

Generically alphabetized processes

To run with this idea, a process $X = (I, O, N) \in \mathbb{D}$ with arbitrary alphabets and an alphabet ordering α determine a generically alphabetized process $\mathcal{F}_{\mathbb{D}}^\alpha(X)$ isomorphic to X such that the n -th generic

symbol $\mathbb{G}^{\circ-1} n$ appears in $\mathcal{T}_{\mathbb{D}\mathbb{D}}^{\alpha}(X)$ wherever the n -th alphabet symbol with respect to α appears in X . Unlike X , this process is worth comparing with a process derived from a circuit because it stands a chance of having the same alphabet as one.

For a process $X = (I, O, N) \in \mathbb{D}$ and an alphabet ordering α , any alphabet symbol $t \in I \cup O$ determines an ordinal

$$((\alpha \uparrow I) \parallel (\alpha \uparrow O))^{-1} t \in \mathbb{N}$$

corresponding to its position in the list $(\alpha \uparrow I) \parallel (\alpha \uparrow O)$, and a generic symbol

$$\mathbb{G}^{\circ-1} ((\alpha \uparrow I) \parallel (\alpha \uparrow O))^{-1} t \in \mathbb{G}.$$

A function mapping any such $t \in I \cup O$ to the corresponding generic symbol but mapping non-members of $I \cup O$ to themselves is expressible as

$$(\lambda t. (\lambda k. \langle \mathbb{G}^{\circ-1} ((\alpha \uparrow I) \parallel (\alpha \uparrow O))^{-1} t, t \rangle_k) \delta_{\emptyset}^{\{t\} \cap (I \cup O)} : \mathbb{T} \cup \mathbb{V} \rightarrow \mathbb{G} \cup \mathbb{V}$$

and one that rewrites every symbol accordingly throughout the process X as

$$(\lambda t. (\lambda k. \langle \mathbb{G}^{\circ-1} ((\alpha \uparrow I) \parallel (\alpha \uparrow O))^{-1} t, t \rangle_k) \delta_{\emptyset}^{\{t\} \cap (I \cup O)})^{\diamond} : \mathbb{D} \rightarrow \mathbb{D}$$

in terms of the notation defined by [Equation 5.11](#). A simple way of defining the transformation $\mathcal{T}_{\mathbb{D}\mathbb{D}}^{\alpha}$ from a process to a generically alphabetized version of itself follows as

$$\mathcal{T}_{\mathbb{D}\mathbb{D}}^{\alpha}(X) = (\lambda(I, O, N). (\lambda t. (\lambda k. \langle \mathbb{G}^{\circ-1} ((\alpha \uparrow I) \parallel (\alpha \uparrow O))^{-1} t, t \rangle_k) \delta_{\emptyset}^{\{t\} \cap (I \cup O)})^{\diamond} X) X. \quad (8.29)$$

8.7.2 More transformations

We can now briefly dispense with the specifications for three transformations $\mathcal{T}_{\mathbb{B}\mathbb{D}}$, $\mathcal{T}_{\mathbb{H}\mathbb{D}}$, and $\mathcal{T}_{\mathbb{L}\mathbb{D}}$ from primitive blocks, hierarchical blocks, and netlists respectively to processes.

A primitive block $X = (I, O, B) \in \mathbb{B}$ with I inputs and O outputs maps to a process $\mathcal{T}_{\mathbb{B}\mathbb{D}}(X) \in \mathbb{D}$ whose input alphabet contains the first I generic symbols and whose output alphabet contains the next O generic symbols when $\mathcal{T}_{\mathbb{B}\mathbb{D}}$ is defined as

$$\mathcal{T}_{\mathbb{B}\mathbb{D}} = \lambda(I, O, B). (\lambda g. B(g \upharpoonright I, g \ll I)) (\mathbb{G}^{\circ-1})^* \iota_{I+O} \quad (8.30)$$

In this way, primitive blocks with similar arities always map to processes with similar alphabets, so only their behavior is left to distinguish them for purposes of comparison.

A function following the same convention for hierarchical blocks $X \in \mathbb{H}$ follows immediately by defining $\mathcal{T}_{\mathbb{H}\mathbb{D}}$ as

$$\mathcal{T}_{\mathbb{H}\mathbb{D}} = \mathcal{T}_{\mathbb{B}\mathbb{D}} \circ \mathcal{T}_{\mathbb{H}\mathbb{B}} \quad (8.31)$$

in terms of $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ as given by [Equation 8.26](#).

For a netlist $X \in \mathbb{L}$, every term $(I, O, (I', O', B)) \in \mathcal{R}(X)$ with a block $(I', O', B) \in \mathbb{B}$ determines a process $B((\mathbb{G}^{\circ-1})^* I, (\mathbb{G}^{\circ-1})^* O) \in \mathbb{D}$ with generic alphabets obtained by applying the semantic function B of the block to the pair of lists of generic symbols whose ordinals are the wire numbers in I and O . Because the wires are numbered globally within a netlist, these processes have alphabets that intersect precisely as needed to effect connections among them in their parallel composition analogous to those of the blocks in the netlist. A result of the form

$$(\mathcal{F} \text{ par}) (\lambda(I, O, (I', O', B)). B((\mathbb{G}^{\circ-1})^* I, (\mathbb{G}^{\circ-1})^* O)) X \in \mathbb{D}$$

therefore would be a process with a generic alphabet corresponding in some way to the netlist X , but not necessarily having minimal alphabet ordinals per convention. We can attend to this detail by generically alphabetizing the result in the following definition of $\mathcal{J}_{\mathbb{L}\mathbb{D}}$

$$\mathcal{J}_{\mathbb{L}\mathbb{D}} = \mathcal{J}_{\mathbb{D}\mathbb{D}}^{\mathbb{G}^{\circ-1}} \circ (\mathcal{F} \text{ par}) \circ (\lambda(I, O, (I', O', B)). B((\mathbb{G}^{\circ-1})^* I, (\mathbb{G}^{\circ-1})^* O))^* \quad (8.32)$$

using $\mathbb{G}^{\circ-1}$ as the alphabet ordering in $\mathcal{J}_{\mathbb{D}\mathbb{D}}^{\mathbb{G}^{\circ-1}}$ as defined by Equation 8.29, which is appropriate because the alphabet is already generic.

8.7.3 Generalized refinement

A refinement relation encompassing circuits represented as blocks, netlists, or processes depends on transforming them all to processes before comparing them. Each of the transformations developed above plays a part in enabling this result. A transformation $\mathcal{J}_{\mathbb{D}}^{\alpha} : \mathbb{L} \cup \mathbb{H} \cup \mathbb{D} \rightarrow \mathbb{D}$ defined with respect to an alphabet ordering $\alpha \in \mathbb{T}^*$

$$\mathcal{J}_{\mathbb{D}}^{\alpha}(X) = \begin{cases} \mathcal{J}_{\mathbb{L}\mathbb{D}} X & \text{if } X \in \mathbb{L} \\ \mathcal{J}_{\mathbb{B}\mathbb{D}} \mathcal{J}_{\mathbb{H}\mathbb{B}} X & \text{if } X \in \mathbb{H} \\ \mathcal{J}_{\mathbb{D}\mathbb{D}}^{\alpha} X & \text{if } X \in \mathbb{D} \end{cases} \quad (8.33)$$

gives rise to a refinement relation applicable to any $X, Y \in \mathbb{L} \cup \mathbb{H} \cup \mathbb{D}$ denoted $X \stackrel{\alpha}{\sqsubseteq} Y$ and read “ X is refined by Y under α ”, or equivalently, “ Y refines X under α ”, where $\alpha \in \mathbb{T}^*$ is an alphabet ordering. This relation is defined as that which satisfies

$$X \stackrel{\alpha}{\sqsubseteq} Y \Leftrightarrow \mathcal{J}_{\mathbb{D}}^{\alpha}(X) \sqsubseteq \mathcal{J}_{\mathbb{D}}^{\alpha}(Y) \quad (8.34)$$

where the ordinary refinement relation \sqsubseteq on \mathbb{D} is given by Equation 7.18. Generalized forms of equivalence and antirefinement under an alphabet ordering are defined similarly. As noted in Section 8.7.1, defining an alphabet ordering is simply a matter of making a list of the alphabet symbols. Whether α is chosen in an *ad hoc* way or according to some convention is left open.

Although this relation is rarely invoked as such in this book after this point, it achieves an important milestone by unifying the relevant semantic models within a common framework, thereby allowing some chance in principle of settling arguments about compatibility or correctness whenever circuits and processes are both mentioned together. It also summarizes in one line the central problem of DI circuit verification.

8.8 Connection patterns

The discussion of the block combinators **R**, **Z**, and **I** up to this point could give the impression that they restrict the ways of connecting the blocks. Connecting the first output terminal of a block X to its last input is expressed $\mathbf{Z}(X)$, and connecting the first output of X to the last input of Y could be expressed $\mathbf{ZR}(X, Y)$, but there is no apparent provision for connecting arbitrarily selected terminals in larger networks. Two remedies are described below. A method supportive of schematic capture in Section 8.8.1 works with explicitly enumerated wire origin and terminus addresses, while a method described in Section 8.8.2 routes a bundle of connections *en masse* according to a specified permutation.

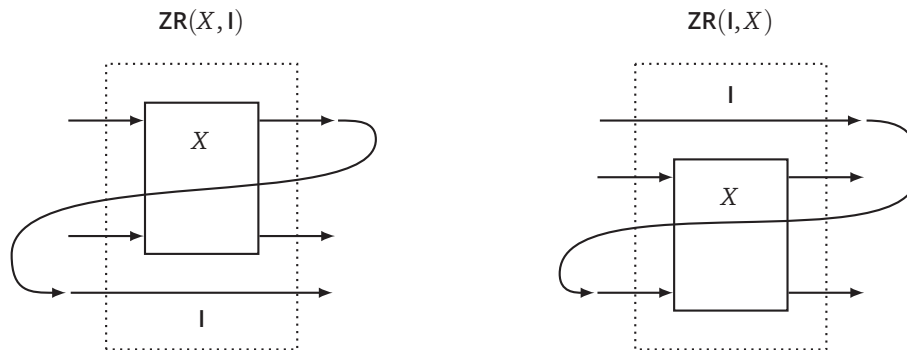


Figure 8.7: At left, $ZR(X, I)$ is equivalent to X with the output terminals rolled up by one position, whereas at right, $ZR(I, X)$ is equivalent to X with the input terminals rolled down by one.

8.8.1 Schematic capture

Although the Z combinator applies only to terminals at the first and last positions on a block, it incurs no loss of generality. Any terminal of a block X can be mapped to another position by some combination of X with R , Z and I . Following a justification of this claim and some suggestions for a workable notation, this section concludes with a discussion of a more systematic approach.

Terminal rotation

The achievement of general connection patterns from the basic block combinators is built on expressions in two simple forms. The expression $ZR(X, I)$ describes a block with the same arity as X and a similar semantics, but with the output terminals in a different order. If $ZR(X, I)$ were used as a replacement for X in a fixed environment, then every time X would have sent an output signal from its first output terminal, $ZR(X, I)$ would send one from its last. Every time X would have sent a signal from an output other than the first, $ZR(X, I)$ would send one from the preceding terminal in the ordering. This behavior is apparent from [Figure 8.7](#) as a direct consequence of the block combinator semantics and the understanding that I acts as a wire. In effect, the second terminal is moved to the first position, and therefore could be connected subsequently to something else by the Z combinator.

A related effect can be inferred from [Figure 8.7](#) for the expression $ZR(I, X)$. In this case, the order of the outputs is unaffected but the input order differs. Whenever a signal is sent to the first input terminal of $ZR(I, X)$, it reacts as X would have reacted if a signal were sent to its last input. Whenever a signal is sent to any input other than the first, it reacts as X would have reacted if a signal were sent to the preceding one in the input terminal ordering. The effect is as if the inputs had been cyclically shifted or “rolled” one position downward in the diagram at right, whereas in the case of $ZR(X, I)$ at left, the outputs are rolled up by one.

The implication for the example of $ZR(X, Y)$ at the beginning of this section is that alternative connection patterns are not at all precluded. To express a connection originating from the second



output of X instead of the first, we can write $\mathbf{ZR}(\mathbf{ZR}(X, I), Y)$. To express it terminating at the penultimate input of Y instead of the last, we could write $\mathbf{ZR}(X, \mathbf{ZR}(I, Y))$. To do both, we could write $\mathbf{ZR}(\mathbf{ZR}(X, I), \mathbf{ZR}(I, Y))$.

There is no need to stop at this point. A connection would be indicated from the third output of X if $\mathbf{ZR}(\mathbf{ZR}(X, I), I)$ were used in place of X in $\mathbf{ZR}(X, Y)$, assuming X has more than two output terminals, and could terminate at the preceding input of Y if $\mathbf{ZR}(I, \mathbf{ZR}(I, Y))$ were used in place of Y . Clearly these expressions can be nested independently to the depth required to achieve a connection between any choice of terminals.

Terminal rotations are required frequently enough to justify a notation encapsulating the equivalent block combinator expressions with less clutter. In the remainder of this section, three variations on this theme are proposed.

Rolling forward The following notation is applicable to express a downward rotation of input terminals by n ,

$$X \downarrow n = (\mathbf{Z} \circ \lambda x. \mathbf{R}(I, x))^n X \quad (8.35)$$

along with this notation for an upward rotation of output terminals by n ,

$$X \uparrow n = (\mathbf{Z} \circ \lambda x. \mathbf{R}(x, I))^n X$$

where $X \in \mathbb{H}$ and n is a natural number. The intended mnemonic significance is that the direction of the arrow indicates that of the rotation, and the side of the arrowhead indicates whether it pertains to inputs or outputs, because blocks are often drawn with the input terminals along the left side and the first input or output terminal at the top. If the displacement n in either of the expressions above is a multiple of the block input or output arity, respectively, then the terminals are effectively rolled back to their original positions and the result is behaviorally equivalent to X .

Using this notation to express a network containing X and Y with the i -th output of X (numbered from zero) connected to the j -th input of Y , we could write

$$\mathbf{ZR}(X \uparrow i, Y \downarrow k)$$

where $j + k + 1$ is the input arity of Y . In this way, the i -th output rolls up to the top position, while the j -th input rolls down to the last.

Rolling backwards If we could roll the outputs down or the inputs up, many networks could be expressed more simply without explicit reference to the arity, and there would be no further need to remember which directions are undefined. Terminal rotations in the other directions can be defined easily like this.

$$\begin{aligned} X \uparrow n &= X \downarrow (\lambda(I, O, B). I - (n \bmod I)) \mathcal{F}_{\mathbb{H}\mathbb{B}} X \\ X \downarrow n &= X \uparrow (\lambda(I, O, B). O - (n \bmod O)) \mathcal{F}_{\mathbb{H}\mathbb{B}} X \end{aligned}$$

That is, to roll the terminals the other direction, we actually roll them in the same direction as before, but by a complementary displacement with respect to the block arity. In this notation, the expression for the network above is at least partly simplified.

$$\mathbf{ZR}(X \uparrow i, Y \uparrow j + 1)$$

Rolling both ways Sometimes it may be useful to rotate both the inputs and the outputs of a block, either in the same or opposite directions, and on rare occasions, by the same displacement. Expressions in these cases might be abbreviated as follows,

$$\begin{aligned} X \Downarrow n &= X \downarrow n \downarrow n \\ X \Uparrow n &= X \downarrow n \uparrow n \\ X \Downarrow n &= X \uparrow n \downarrow n \\ X \Uparrow n &= X \uparrow n \uparrow n \end{aligned}$$

which is to say that the displacement need only be written once.

Wire wrapping

Useful though it may be to connect any two terminals of our choice between blocks X and Y as shown in the previous section, it is obviously not enough, and the difficulty escalates even in the next simplest case. For two specified connections between two blocks, an expression of the following form might be needed.

$$\mathbf{Z}((\mathbf{Z}\mathbf{R}(X \uparrow a, Y \uparrow b)) \uparrow c \uparrow d)$$

The inner displacements a and b are much as before, but the correct values of c and d depend on a , b , and the arities of X and Y , all of which interact during the initial rotations to displace the terminals involved in the second connection from their initial positions.

A uniform approach inspired by a wire wrapping work flow stands a better chance of coping with complex connection patterns than continuing in this vein [189]. To follow this methodology for the circuit in Figure 8.5, we would list the blocks *foo*, *bar* and *baz* in some arbitrary order, such as $X = (\mathcal{F}\mathbf{R})\langle \text{foo}, \text{bar}, \text{baz} \rangle$, and then number the input and output terminals globally in separate consecutive sequences from zero as shown in Figure 8.8. That is, if one block precedes another in the list, then the terminals of the preceding block must be assigned lower numbers, and any two terminals of the same type on the same block must be numbered consistently with their relative order on the block. Next we would enumerate the wires in the circuit with each identified by its source output terminal number and destination input terminal number. In the current example, they are (0, 3), (1, 5), (2, 2), (3, 6), and (4, 1) by inspection of Figure 8.5 and Figure 8.8.

This technique is applicable more generally and worth automating. Let $X \dashv w$ denote the block derived from X by making all connections indicated by a bijective relation $w \in \mathcal{P}(\mathbb{N} \times \mathbb{N})$. How hard can its specification be in terms of block combinators and terminal rotations? If w contained only one pair (o, i) , then we could write

$$\mathbf{Z}(X \uparrow i + 1 \uparrow o)$$

to make the connection from output number o to input number i and then

$$z = X \dashv \{(o, i)\} = (\mathbf{Z}(X \uparrow i + 1 \uparrow o)) \downarrow i \downarrow o$$

to put the rest of the terminals back into their original order. If another connection $(o', i') \in w$ were also indicated, it would not be correct to continue by writing

$$(\mathbf{Z}(z \uparrow i' + 1 \uparrow o')) \downarrow i' \downarrow o'$$

because the terminal numbers i' and o' might not be applicable to the block z left by the first connection. If o' is greater than o , then the output terminal numbered o' on X corresponds to the

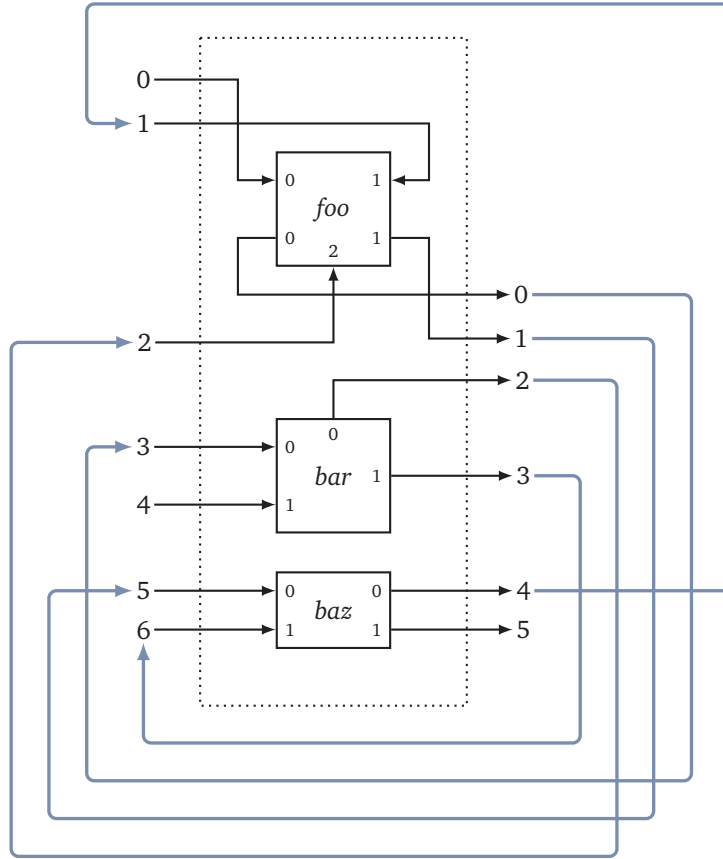


Figure 8.8: The expression $((\mathcal{F}\mathbf{R})\langle foo, bar, baz \rangle) \dashv \{(0, 3), (1, 5), (2, 2), (3, 6), (4, 1)\}$ encodes the network in Figure 8.5 as a member of \mathbb{H} and is captured from the schematic by listing it as shown.

one numbered $o' - 1$ on z . We might say in general that it corresponds to the output terminal numbered $u_4(o, \mathcal{D}(w))$ o' in terms of a function $u_4 : \mathbb{N} \times \mathcal{P}(\mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ given by

$$u_4 = \lambda(o, d). (\lambda s. \lambda o'. o' - \delta_{d^o o'}^{1+s^o o'}) (d - \{o\})$$

which is to say the predecessor of o' whenever the ordinal of o' with respect to $\mathcal{D}(w)$ differs from its ordinal with respect to $\mathcal{D}(w) - \{o\}$. Similarly, input terminal number i' on X corresponds to input terminal number $u_4(i, \mathcal{R}(w))$ i' on z , and hence the pair of terminal numbers (o', i') on X corresponds to the pair $(u_5(o, i) w) (o', i')$ for a function

$$u_5 : (\mathbb{N} \times \mathbb{N}) \rightarrow (\mathcal{P}(\mathbb{N} \times \mathbb{N}) \rightarrow ((\mathbb{N} \times \mathbb{N}) \rightarrow (\mathbb{N} \times \mathbb{N})))$$

given by

$$u_5 = \lambda(o, i). \lambda w. \lambda(o', i'). (u_4(o, \mathcal{D}(w)) o', u_4(i, \mathcal{R}(w)) i').$$

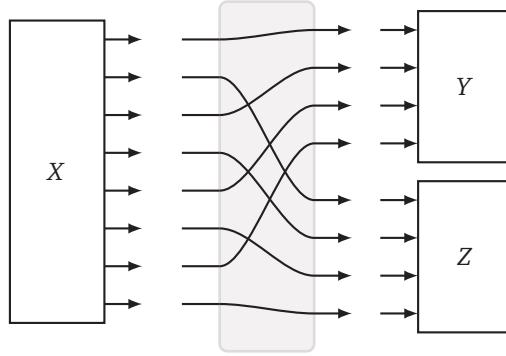


Figure 8.9: Interposing a permutation network between X and the parallel combination of Y and Z separates the even from the odd outputs.

on z . This observation suggests a strategy of making a list $w^{\circ-1} \in (\mathbb{N} \times \mathbb{N})^*$ of the members of w and transforming it to a list

$$c = (\mathcal{F}_\epsilon \lambda(h, t). (((u_5 h) \mathcal{R}(h : t))^* t) \parallel \langle h \rangle) w^{\circ-1}$$

whose last term $c_{|c|-1}$ is numbered relative to the original block X but whose other terms are numbered relative to a block derived from X by making every connection indicated by their succeeding terms. Provided we make the connection indicated by each term in c in an order starting from the last one and working backwards as

$$(\mathcal{F}_X \lambda((o, i), z). (\mathbf{Z}(z \uparrow i + 1 \uparrow o)) \downarrow i \downarrow o) c$$

we should have a correct result for $X \dashv w$, and therefore the following definition.

$$X \dashv w = (\mathcal{F}_X \lambda((o, i), z). (\mathbf{Z}(z \uparrow i + 1 \uparrow o)) \downarrow i \downarrow o) (\mathcal{F}_\epsilon \lambda(h, t). (((u_5 h) \mathcal{R}(h : t))^* t) \parallel \langle h \rangle) w^{\circ-1}$$

8.8.2 Permutations

The wire wrap operator \dashv defined above demonstrates the universality of the block combinators with regard to network connection patterns, and is well suited to automated schematic capture for small manual designs, but more parsimonious descriptions are necessary for coping effectively with realistic requirements. Often a circuit is better described in aggregate terms than on the level of individual connections, particularly when there are symmetric or repetitive structures involved. Connecting two blocks by a bus is one obvious example. Connecting each terminal on a block to one of an array of blocks is another.

These examples point to a general need for organizing the connections into meaningfully related groups. Maybe some non-contiguous set of output terminals on a block is to be connected to one destination, and the rest to another, as in Figure 8.9, where the even outputs from the block X at the left are meant for the block Y at the upper right, and the odd outputs for Z at the lower right. Despite its conceptual simplicity, this network would be difficult to express transparently by block combinators alone, even with the use of the wire wrap or terminal rotation operators.



Figure 8.10: Left, the permutation network described by $\langle 2, 0, 1 \rangle$ acknowledges input 0 with output 2, input 1 with output 0, and input 2 with output 1, contrary to that of $\langle 2, 0, 1 \rangle^{-1}$, right.

However, there is quite a simple way to describe the network in Figure 8.9 and many others like it, which is to treat the shaded region of the figure as a block in itself even though it has no internal components, and let it encapsulate most of the complexity. A block having the same number of output terminals as inputs, with nothing but a wire from each input to one of the outputs, is called a **permutation network** hereafter. Any desired rearrangement of the terminal order on a block can be expressed as a combination of that block with a suitably chosen permutation network, and any permutation network itself can be described completely by a permutation (Section 8.1.5). Useful permutations for connection patterns exhibiting some form of regular structure are likely to have succinct descriptions in terms of familiar list operations such as shuffling and transposition (Section 8.1.7). For example, the permutation network in Figure 8.9 is described by $t_8 \times 2$.

Permutation networks always can be built manually from block combinators with sufficient effort, but it is easier to generate them automatically to order. This section starts with a discussion of a simple recursive algorithm to derive a block combinator representation of a permutation network implementing any given permutation, to be followed by some notational suggestions for the use of permutation networks in buses and terminal rotations.

Permutation network synthesis

Permutation networks were studied extensively and perhaps exhaustively following the seminal contribution of [298], which specified an algorithm to construct a multi-stage interconnection network implementing any given permutation of length 2^n using only the two possible binary permutation networks as building blocks. Whereas the early work was motivated by telephone switching networks [22], subsequent research was motivated by shared-memory multiprocessor architectures, focusing on optimization [132] and generalization [52] of the class of synthesizable permutation networks, improvements to the synthesis algorithm [162], complexity analysis [6], and dynamic reconfigurability [161]. In this context, our present agenda could be classified as a restriction to non-reconfigurable (*i.e.*, “hard wired”) single-stage interconnection networks, eschewing a standard cell approach, and hence momentarily ignoring matters of placement, latency, and whatever else may threaten to make it the least bit difficult except for a modest concession to efficiency noted below.

A permutation network is specified by a bijective list $s \in \mathbb{N}^*$ with the understanding that a signal imparted to the i -th input is acknowledged on the s_i -th output as Figure 8.10 illustrates. (The inverse of this convention would also work, but we must choose one and stick to it.) Our sole aim is to construct a new block combinator $\mathbf{P} : \mathbb{N}^* \rightarrow \mathbb{H}$ whereby $\mathbf{P}(s)$ can denote a permutation network described by s , with \mathbf{P} defined ultimately in terms of the basic block combinators \mathbf{R} , \mathbf{Z} and \mathbf{I} .

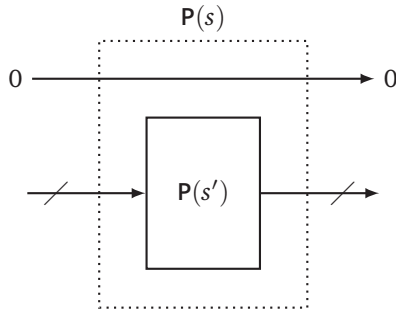


Figure 8.11: A permutation network $P(s)$ where $s_0 = 0$ is of the form $TP(s')$, which is a wire in parallel with a permutation network $P(s')$ with $|s'| = |s| - 1$.

One way of defining the permutation network combinator would be in terms of the wire wrap operator defined in Section 8.8.1

$$P = \lambda s. !^{2|s|} \dashv (\mu \lambda n. (n, |s| + s_n)) \mathcal{D}(s) \tag{8.36}$$

but this approach is inefficient in both the size of the block combinator expressions it generates and the time needed to generate them. A bit of further inquiry leads to a way of generating efficient permutation networks directly with less effort.

This task is aided by something like a change of variables first. In place of R and Z , let us temporarily define two alternative block combinators T and S as follows.

$$T = \lambda x. R(I, x) \tag{8.37}$$

$$S = Z \circ T \tag{8.38}$$

The T combinator has the effect of preceding its argument with a wire in parallel, while $S(X)$ is equivalent to $X \downarrow 1$, a downward rotation of the inputs by one position. (See Figure 8.7 and Equation 8.35.) For example, the network at the left in Figure 8.10 can be expressed as $STTI$, or more briefly ST^2I , which expands to $ZR(I, R(I, R(I, I)))$. Clearly any block combinator expression featuring T , S and I can be rewritten in terms of R , Z and I .

Aside from trivial cases like $P\langle 0 \rangle = I$, finding an algorithm to compute $P(s)$ for an arbitrary bijective $s \in \mathbb{N}^*$ could be a tough nut to crack. Our one chance to catch a break is by noting that if s_0 is 0, then the first input of the permutation network must be wired directly to the first output however tangled the rest of it may be. This observation implies a form of $TP(s')$ for $P(s)$, meaning $R(I, P(s'))$ by Equation 8.37, because it is a wire in parallel with a smaller permutation network $P(s')$ as shown in Figure 8.11.⁴ The permutation s' does not reflect the wire described by $s_0 = 0$ but is similar to s otherwise, in that each item of s' is the corresponding item of s offset by 1. Hence we can infer the relationship

$$P(s) = (\lambda s'. TP(s')) (\lambda i. i - 1)^* (s \ll 1) \tag{8.39}$$

⁴A wire with a slash through it in a circuit schematic represents a number of wires in parallel.

at least for the special case of $s_0 = 0$. This relationship is helpful for computing $\mathbf{P}(s)$ only if $\mathbf{P}(s')$ is known, which can be found recursively only if every subsequent s'_0 is 0. The effect is to limit the scope of this result somewhat disappointingly to identity permutations ι_n .

The key to generalizing Equation 8.39 to an arbitrary permutation s is that any permutation network $\mathbf{P}(s)$ can be obtained by an input terminal rotation from some permutation network of the form $\mathbf{TP}(s')$ with $|s'| = |s| - 1$. The problem is to choose s' and a rotational displacement h so that $\mathbf{S}^h \mathbf{TP}(s')$ becomes the desired permutation network $\mathbf{P}(s)$. Fortunately, it is easy to see that the only choice for the displacement h is $s^{-1}(0)$, because for the network $\mathbf{TP}(s')$, input terminal 0 is acknowledged by output 0, so only after rolling down its inputs by $s^{-1}(0)$ can we have input $s^{-1}(0)$ of the resulting network acknowledged by output 0 as required for $\mathbf{P}(s)$. Hence the problem reduces to that of solving the equation

$$\mathbf{P}(s) = \mathbf{S}^h \mathbf{TP}(s') \quad (8.40)$$

for the permutation s' describing a network that morphs into $\mathbf{P}(s)$ when a wire is placed in parallel with it and its inputs are rotated downward by $h = s^{-1}(0)$.

Solving Equation 8.40 is not as hopeless as it looks if we observe that the \mathbf{S} , \mathbf{T} , and \mathbf{P} combinators satisfy certain identities, which can be confirmed by drawing diagrams similar to the one in Figure 8.11.

$$\begin{aligned} \mathbf{TP}(x) &= \mathbf{P}(0 : (\lambda i. i + 1)^* x) \\ \mathbf{SP}(x) &= \mathbf{P}(x_{|x|-1} : (x \upharpoonright |x| - 1)) \end{aligned}$$

For any natural number $n \leq |x|$, the latter generalizes to

$$\mathbf{S}^n \mathbf{P}(x) = \mathbf{P}((x \ll |x| - n) \parallel (x \upharpoonright |x| - n))$$

which is to say that applying \mathbf{S} to a permutation network n times is equivalent to starting with a permutation rotated by n indices from the original. With that, we have

$$\begin{aligned} \mathbf{P}(s) &= \mathbf{S}^h \mathbf{TP}(s') \\ &= \mathbf{S}^h \mathbf{P}(0 : (\lambda i. i + 1)^* s') \\ &= \mathbf{P}((\lambda q. (q \ll |q| - h) \parallel (q \upharpoonright |q| - h)) 0 : (\lambda i. i + 1)^* s'). \end{aligned}$$

Equating the permutations from both sides yields

$$\begin{aligned} (\lambda q. (q \ll |q| - h) \parallel (q \upharpoonright |q| - h)) 0 : (\lambda i. i + 1)^* s' &= s \\ 0 : (\lambda i. i + 1)^* s' &= (s \ll h) \parallel (s \upharpoonright h) \\ (\lambda i. i + 1)^* s' &= (s \ll h + 1) \parallel (s \upharpoonright h) \\ s' &= (\lambda i. i - 1)^* ((s \ll h + 1) \parallel (s \upharpoonright h)) \end{aligned}$$

which agrees with Equation 8.39 when h is equal to 0.

This result makes $\mathbf{P}(s)$ expressible as a recurrence for any bijective $s \in \mathbb{N}^*$ subject to a minor additional provision for the degenerate case of $s = \epsilon$.

$$\mathbf{P}(s) = \begin{cases} \langle \mathbf{ZI}, \mathbf{I} \rangle_{|s|} & \text{if } |s| \leq 1 \\ (\lambda h. \mathbf{S}^h \mathbf{TP}((\lambda i. i - 1)^* ((s \ll h + 1) \parallel (s \upharpoonright h)))) s^{-1}(0) & \text{otherwise} \end{cases} \quad (8.41)$$

Here, \mathbf{ZI} represents the nullary permutation network because it has no inputs and no outputs. Some examples of permutation networks generated according to Equation 8.41 for permutations up to length 4 are shown in Table 8.1.

s	$P(s)$	s	$P(s)$	s	$P(s)$
ϵ	ZI	$\langle 0, 2, 1, 3 \rangle$	TSTSTI	$\langle 2, 1, 0, 3 \rangle$	S^2TS^2TSTI
$\langle 0 \rangle$	I	$\langle 0, 2, 3, 1 \rangle$	TS^2T^2I	$\langle 2, 1, 3, 0 \rangle$	$S^3TSTSTI$
$\langle 0, 1 \rangle$	TI	$\langle 0, 3, 1, 2 \rangle$	TST^2I	$\langle 2, 3, 0, 1 \rangle$	S^2T^3I
$\langle 1, 0 \rangle$	STI	$\langle 0, 3, 2, 1 \rangle$	TS^2TSTI	$\langle 2, 3, 1, 0 \rangle$	$S^3TS^2T^2I$
$\langle 0, 1, 2 \rangle$	T^2I	$\langle 1, 0, 2, 3 \rangle$	STS^2T^2I	$\langle 3, 0, 1, 2 \rangle$	ST^3I
$\langle 0, 2, 1 \rangle$	TSTI	$\langle 1, 0, 3, 2 \rangle$	STS^2TSTI	$\langle 3, 0, 2, 1 \rangle$	STSTSTI
$\langle 1, 0, 2 \rangle$	STSTI	$\langle 1, 2, 0, 3 \rangle$	S^2TST^2I	$\langle 3, 1, 0, 2 \rangle$	$S^2TS^2T^2I$
$\langle 1, 2, 0 \rangle$	S^2T^2I	$\langle 1, 2, 3, 0 \rangle$	S^3T^3I	$\langle 3, 1, 2, 0 \rangle$	S^3TST^2I
$\langle 2, 0, 1 \rangle$	ST^2I	$\langle 1, 3, 0, 2 \rangle$	$S^2TSTSTI$	$\langle 3, 2, 0, 1 \rangle$	S^2T^2STI
$\langle 2, 1, 0 \rangle$	S^2TSTI	$\langle 1, 3, 2, 0 \rangle$	S^3T^2STI	$\langle 3, 2, 1, 0 \rangle$	S^3TS^2TSTI
$\langle 0, 1, 2, 3 \rangle$	T^3I	$\langle 2, 0, 1, 3 \rangle$	ST^2STI		
$\langle 0, 1, 3, 2 \rangle$	T^2STI	$\langle 2, 0, 3, 1 \rangle$	STST 2I		

Table 8.1: networks generated by Equation 8.41 for all permutations up to length 4 using the T and S combinators defined respectively in Equation 8.37 and Equation 8.38

Permutation bus notation

One way of using a permutation network is as a bus to connect two blocks together. If a block X has n outputs and a block Y has n inputs, then a permutation network $P(s)$ with $|s| = n$ could interface between them to form the block $Z^nR(X, Z^nR(P(s), Y))$. A more transparent and succinct notation for this expression would be the following

$$X \xrightarrow{s} Y$$

so that for example the network in Figure 8.9 could be expressed

$$X \xrightarrow{t_8 \times 2} R(Y, Z).$$

However, the expression $Z^nR(X, Z^nR(P(s), Y))$ is not a completely satisfactory definition for $X \xrightarrow{s} Y$ because it yields confusing and unintuitive results when the arities of X and Y do not precisely match $|s|$, which could happen if X and Y have other inputs or outputs than those on the bus.

The meaning of $X \xrightarrow{s} Y$ with non-matching arities would be less confusing if it were as shown in Figure 8.12, because then it would be safe to assume that the inputs and outputs of the combined block were ordered consistently with those of the block $(\mathcal{F} R) \langle X, P(s), Y \rangle$, which is like $X \xrightarrow{s} Y$ without the connections. That is, the first outputs from the block are the unconnected outputs from X , if any, the next are from $P(s)$, and the remainder are from Y . Similarly the inputs to X are first, but any unused inputs to $P(s)$ come next, and unused inputs to Y follow.

An expression that captures the relationships in Figure 8.12 would have to depend on the number of connections C_{Xs} made from X to $P(s)$, and the number of connections C_{sY} from $P(s)$ to Y . Neither of these can be greater than $|s|$, nor can C_{Xs} or C_{sY} exceed the output and input arities of X and Y respectively. To keep track of these conditions, let C_{Xs} and C_{sY} be defined as

$$C_{Xs} = \min\{O_X, |s|\} \quad (8.42)$$

$$C_{sY} = \min\{|s|, I_Y\} \quad (8.43)$$

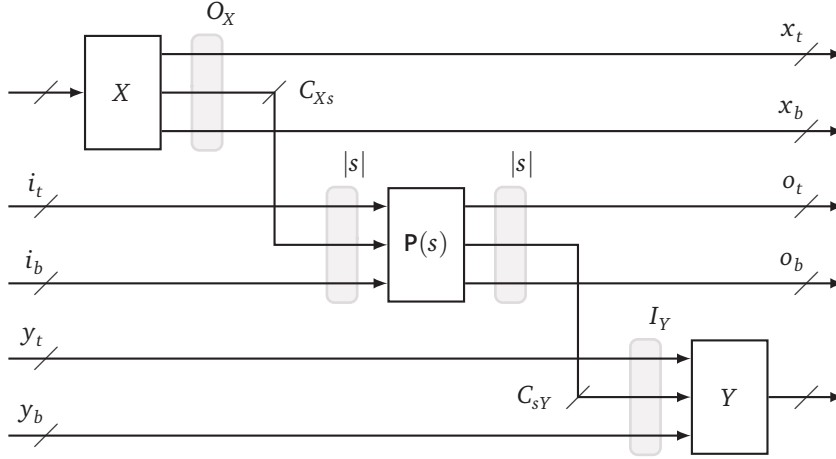


Figure 8.12: Bus width labels indicate the number of unused inputs and outputs at each end of blocks X , $P(s)$, and Y when not all arities in $X \xrightarrow{s} Y$ match.

where the output and input arities of X and Y are defined as follows

$$O_X = (\lambda(I, O, B). O) \mathcal{F}_{\text{HIB}} X \quad (8.44)$$

$$I_Y = (\lambda(I, O, B). I) \mathcal{F}_{\text{HIB}} Y \quad (8.45)$$

using the transformation \mathcal{F}_{HIB} given by Equation 8.26.

As the figure also suggests, let us further restrict attention to permutation buses connected to contiguous ranges of terminals on their source and destination blocks rather than to arbitrary subsets thereof. Then in general there could be x_t unused output terminals on X , followed by C_{Xs} connected terminals, followed by another x_b unused outputs satisfying $x_t + C_{Xs} + x_b = O_X$, assuming $|s| < O_X$ holds. Alternatively, if O_X is less than $|s|$, then all of the output terminals on X are connected, but some of the inputs on $P(s)$ are not. Allowing only a contiguous range connected inputs to $P(s)$, we have possibly i_t unused inputs, followed by C_{Xs} connected inputs, followed by i_b unused inputs to $P(s)$, where $i_t + C_{Xs} + i_b = |s|$. Similar conventions apply to the permutation network outputs and the inputs to Y , which are grouped into ranges of size o_t , o_b , y_t and y_b as shown, with $o_t + C_{sY} + o_b = |s|$ and $y_t + C_{sY} + y_b = I_Y$.

A pattern that appears twice in Figure 8.12 features some number of connections from one block to another, excluding some number of leading outputs on the former and some number of trailing inputs on the latter. To describe this pattern in general, let a pair of blocks x and y require c connections from x to y excluding the first t outputs from x and the last b inputs to y . Then the combination

$$\mathbf{Z}^c(\mathbf{R}(x, y) \uparrow t \downarrow b)$$

contains the required connections. Restoring the remaining unconnected terminals to their original relative order on the result can be accomplished by undoing the terminal rotations above.

$$(\mathbf{Z}^c(\mathbf{R}(x, y) \uparrow t \downarrow b)) \downarrow t \uparrow b$$

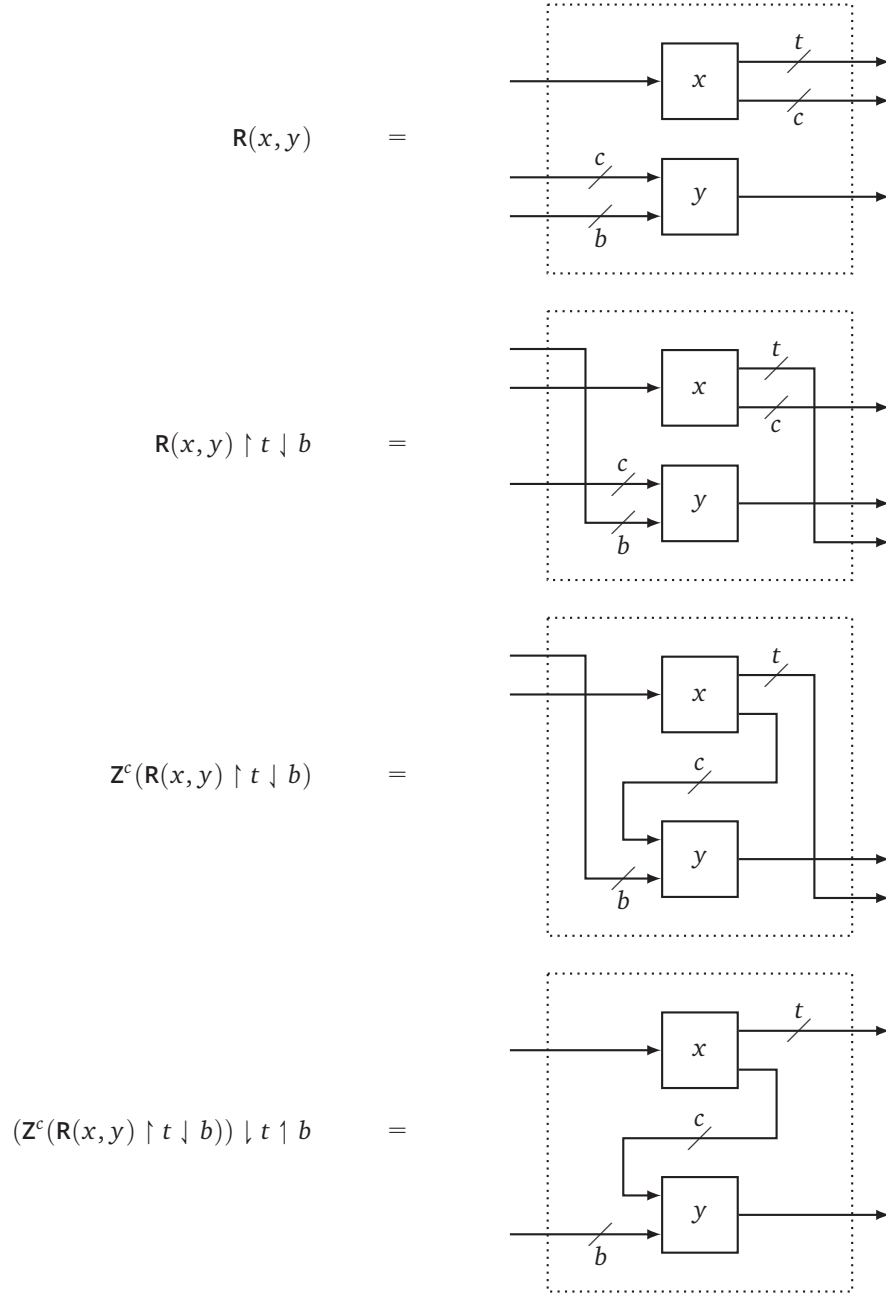


Figure 8.13: how to make c connections from x to y excluding the first t outputs of x and the last b inputs of y without changing the order of the excluded terminals

These operations are illustrated in [Figure 8.13](#). A second order function \dot{N} parameterized by natural numbers c , t , and b maps a pair of blocks (x, y) to a result in this pattern.

$$\dot{N}(c, t, b) = \lambda(x, y). (\mathbf{Z}^c(\mathbf{R}(x, y) \uparrow t \downarrow b)) \downarrow t \uparrow b \quad (8.46)$$

Using this function, we can specify a general form of $X \xrightarrow{s} Y$ by reading it from [Figure 8.12](#). First we have the partial result

$$z = \dot{N}(C_{sY}, o_t, y_b) (\mathbf{P}(s), Y) \quad (8.47)$$

which is a block with $i_t + C_{Xs} + i_b + y_t + y_b$ inputs. To connect the right C_{Xs} outputs from X to the right C_{Xs} inputs on z , we need to avoid the first x_t outputs on X and the last $i_b + y_t + y_b$ inputs on z , which is easily done as follows.

$$z' = \dot{N}(C_{Xs}, x_t, i_b + y_t + y_b) (X, z) \quad (8.48)$$

The only remaining problem is to choose the parameters shown in [Figure 8.12](#). Fortunately x_b , i_t , and o_b are not explicitly needed in the formula, and for a given X , Y , and s , the value of y_b is fully determined by y_t . This process of elimination leaves x_t , y_t , i_b , and o_t . The latter two are non-zero only when $|s|$ is greater than O_X or I_Y respectively. These cases would not usually occur in practice for appropriate choices of $|s|$, so it is not overly restrictive to opt for a fixed convention regarding i_b and o_t .

$$i_b = \lfloor (|s| - C_{Xs})/2 \rfloor \quad (8.49)$$

$$o_t = \lfloor (|s| - C_{sY})/2 \rfloor \quad (8.50)$$

These definitions imply that if the permutation network is too big for the blocks it interfaces, then only the middle inputs and outputs of the permutation network are connected. If there is an odd number of unconnected inputs or unconnected outputs to the permutation network, then the extra one is at the end. We retain only x_t and y_t as free parameters in light of a few useful ways of varying them to be noted shortly.

To summarize everything from [Equation 8.42](#) up to this point, we can define a second order function based on [Equation 8.47](#) and [Equation 8.48](#) in the style of [Equation 8.46](#), but now parameterized by a permutation s and bus widths x_t and y_t , that takes a pair of blocks X and Y to the configuration shown in [Figure 8.12](#).

$$\ddot{N}(s, x_t, y_t) = \lambda(X, Y). (\lambda y_b. \dot{N}(C_{Xs}, x_t, i_b + y_t + y_b) (X, \dot{N}(C_{sY}, o_t, y_b) (\mathbf{P}(s), Y))) I_Y - y_t - C_{sY}$$

Here, C_{Xs} and C_{sY} are given by [Equation 8.42](#) and [Equation 8.43](#), I_Y is given by [Equation 8.45](#), and i_b and o_t are given by [Equation 8.49](#) and [Equation 8.50](#) with respect to the formal parameters X and Y .

If $X \xrightarrow{s} Y$ is defined to utilize only the middle outputs of X and the middle inputs of Y when there are insufficiently many permutation bus lines to connect all of them, it can be specified as follows (cf. [Equation 8.49](#) and [Equation 8.50](#)).

$$X \xrightarrow{s} Y = \ddot{N}(s, \lfloor (O_X - C_{Xs})/2 \rfloor, \lfloor (I_Y - C_{sY})/2 \rfloor) (X, Y) \quad (8.51)$$

However, there are other potentially useful alternatives, such as interfacing the first $|s|$ outputs from X with the first $|s|$ inputs of Y , or the first $|s|$ outputs with the last $|s|$ inputs, or the last $|s|$ with the

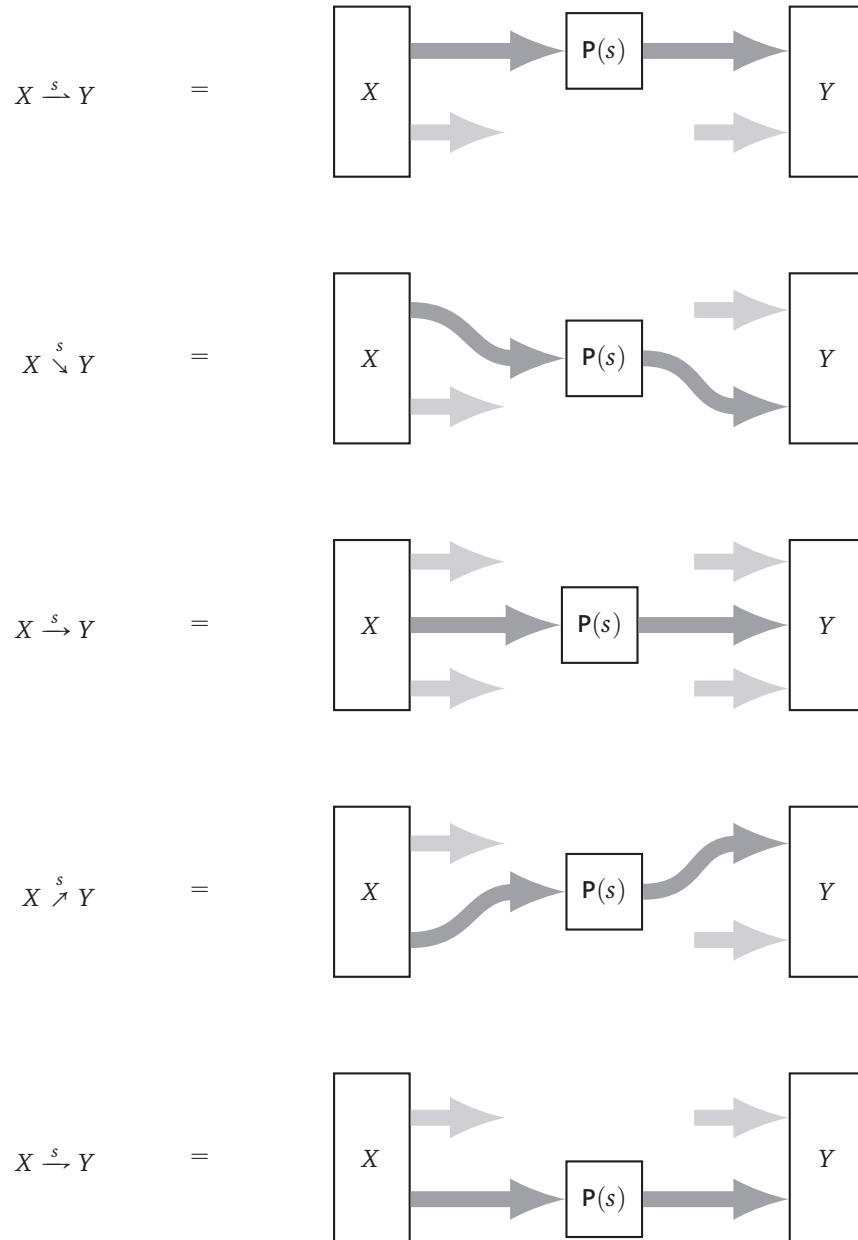


Figure 8.14: A permutation network $P(s)$ of lesser arity than blocks X and Y allows at least five memorable ways to interface them depending on which groups of terminals are left unused.

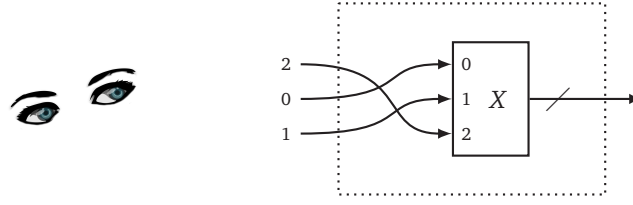


Figure 8.15: $\langle 2, 0, 1 \rangle \times X$ denotes a block X with its inputs permuted by $\langle 2, 0, 1 \rangle$. (cf. Figure 8.10)

last $|s|$, or the last with the first. These possibilities are illustrated in Figure 8.14. We can retain the flexibility to choose any of these configurations by distinguishing among them notationally and in their definitions by the second and third parameters to \ddot{N} .

$$\begin{aligned} X \xrightarrow{s} Y &= \ddot{N}(s, 0, 0)(X, Y) & X \overset{s}{\searrow} Y &= \ddot{N}(s, 0, I_Y - C_{sY})(X, Y) \\ X \xrightarrow{s} Y &= \ddot{N}(s, O_X - C_{Xs}, I_Y - C_{sY})(X, Y) & X \overset{s}{\nearrow} Y &= \ddot{N}(s, O_X - C_{Xs}, 0)(X, Y) \end{aligned}$$

8.8.3 Generalized terminal rotations

In addition to interfacing between two blocks, another use for a permutation network is to express an arbitrary reordering of the terminals on a single block. Three alternatives are discussed in this section, which are general input terminal permutations, general output terminal permutations, and a family of specialized terminal permutations suitable mainly for repetitive arrays of blocks.

Input permutations

An expression $s \times X$, denoting the block X with its inputs reordered according to a permutation s , can be defined in a way that covers all of the edge cases due to mismatched arities by leveraging the permutation bus operator defined above.

$$s \times X = \mathbf{P}(s) \overset{\iota_{|s|}}{\longrightarrow} X \quad (8.52)$$

The identity permutation $\iota_{|s|}$ is used to connect the permutation network $\mathbf{P}(s)$ to X with no further reordering. For example, if X is a block with three inputs, then

$$\langle 2, 0, 1 \rangle \times X$$

is made from X by putting it inside a box with three exposed terminals, and wiring the first input terminal on the box internally to input number 2 on X , the next terminal on the box to input number 0 on X , and the last on the box to terminal number 1 on X . This operation is illustrated in Figure 8.15.

Output permutations

A complementary operation is to permute the outputs of a block. The notation $X \times s$ refers to a block X with its output terminals permuted by s . It is most natural to define it as follows,

$$X \times s = X \overset{\iota_{|s|}}{\longrightarrow} \mathbf{P}(s^{-1}) \quad (8.53)$$

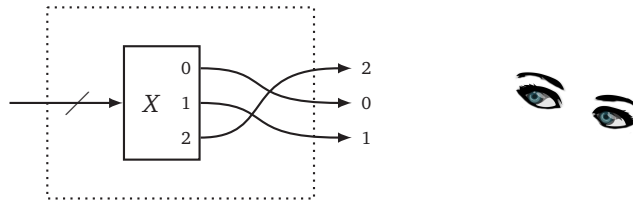


Figure 8.16: $X \times \langle 2, 0, 1 \rangle$ denotes a block X with its outputs permuted by $\langle 2, 0, 1 \rangle$, which involves the permutation network $P\langle 2, 0, 1 \rangle^{-1}$ (cf. Figure 8.15 and Figure 8.10).

using the inverse of s rather than s itself as in the case of input terminal permutations. This operation is illustrated in Figure 8.16.

The reason for using s^{-1} in Equation 8.53 but only s in Equation 8.52 is to enable a memorable rule of thumb for working with these operators: input permutation numbers describe where the wires go, and output permutation numbers whence they come. The mnemonic device is to envision terminal numbers written on the block X as shown in Figure 8.15 and Figure 8.16, and each wire as a fiber optic camera or borescope focused on the number of the terminal it reaches. Each terminal number is displayed externally at the point where the wire crosses the outer box boundary, so that the permutation supplied as the operand to \times or \otimes is visible to an observer as the sequence of numbers thus displayed. This technique applies to both input and output terminal permutations.

Array oriented permutations

A shorter notation for a particularly useful class of terminal permutations is helpful when blocks form repetitive arrays. The situation is illustrated in Figure 8.17, which shows a count of c similar blocks $x_1 \dots x_c$, each with m output terminals, for a total of cm outputs from the aggregate $(\mathcal{F} \mathbf{R}) \langle x_1 \dots x_c \rangle$. The first n outputs from each block x_i are segregated into a single bus of width cn shown at the top, and the remaining $c(m - n)$ output lines are routed to an alternate bus shown below. Perhaps the latter $m - n$ lines from each block x_i carry the data resulting from some calculation performed by x_i , while the first n carry status signals having some other interpretation. In any case, there is bound to be some permutation s for which $X \times s$ captures this pattern precisely, but it would be more convenient to abbreviate the desired network simply as $X \Gamma_m^n$.

A definition for $X \Gamma_m^n$ can be sought in terms of the permutation s whereby $X \times s = X \Gamma_m^n$. To find s , we start with the identity permutation ι_{cm} because at least it has the right length. The transpose

$$a = \iota_{cm} \times c = \langle 0, m, 2m \dots (c - 1)m \rangle \parallel \langle 1, m + 1, 2m + 1 \dots (c - 1)m + 1 \rangle \parallel \dots \quad (8.54)$$

gets further toward the desired result by reordering it conceptually as a concatenation of sublists of length c (cf. Figure 8.1), such that the i -th term of the k -th sublist is the index of the k -th output from the block x_i . Hence the truncation $a \upharpoonright cn$ would contain exactly the indices of the first n outputs from every block. Transforming $a \upharpoonright cn$ from a concatenation of n lists of length c to c lists of length n would put the indices back in ascending order, and is accomplished by another transpose operation.

$$a \upharpoonright cn \times n$$

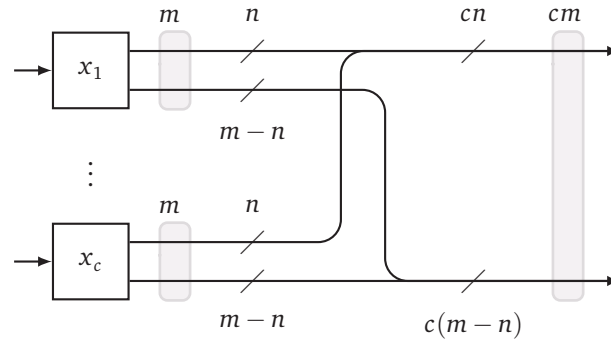


Figure 8.17: For a block $X = (\mathcal{F} \mathbf{R}) \langle x_1 \dots x_c \rangle$ with m outputs from each x_i and cm outputs in total, $X \Gamma_m^n$ partitions the outputs into groups of m and routes the first n of each group to the top.

Similarly, dropping the first cn items of a and transposing the result by $m-n$ yields the list of indices of all but the first n outputs from each block in the correct order.

$$a \ll cn \times m - n$$

The concatenation of these two lists therefore should be the permutation s needed to reorder all of the terminals.

$$s = (a \uparrow cn \times n) \parallel (a \ll cn \times m - n)$$

To express this result as a function of the parameters c , n , and m given originally, we may write

$$A(c, n, m) = (\lambda a. (a \uparrow cn \times n) \parallel (a \ll cn \times m - n)) (t_{cm} \times c).$$

For a spot check of the permutation that separates the first three of ten outputs from each of two blocks, let $c = 2$, $n = 3$, and $m = 10$.

$$A(2, 3, 10) = \langle \overbrace{0, 1, 2}^n, \overbrace{10, 11, 12}^n, \overbrace{3, 4, 5, 6, 7, 8, 9}^{m-n}, \overbrace{13, 14, 15, 16, 17, 18, 19}^{m-n} \rangle$$

If X has O_X output terminals per Equation 8.44, then the number c of blocks can be inferred as O_X/m assuming they are divisible. This result leads to the following definition for $X \Gamma_m^n$.

$$X \Gamma_m^n = X \times A(O_X/m, n, m)$$

There is no need to stop here without throwing in a few other variations. A similar operation for input terminals can be defined as follows, where I_X is given by $(\lambda(I, O, B). I) \mathcal{J}_{\text{HBB}} X$.

$$X \leftarrow_m^n = A(I_X/m, n, m) \times X$$

Finally, alternate versions of these operators that move the first n of every m connections to the bottom instead of the top can be defined in the obvious way.

$$X \leftarrow_m^n = X \leftarrow_m^n \uparrow n I_X/m \quad (8.55)$$

$$X \rightarrow_m^n = X \Gamma_m^n \downarrow n O_X/m \quad (8.56)$$

8.9 Repetitive structures

The assortment of techniques proposed in the previous section to specify connection patterns would be complemented by a similarly expressive means of enumerating the blocks they connect if there were one, but therein lies the rub. The repetitive array structures and cascades essential to many designs are the least of our worries, because anything of interest is far more complicated. Nevertheless, this chapter concludes briefly with a few notational suggestions to the extent such things may be useful, as indeed they are when we have to eat our own dog food starting in Part III.

8.9.1 Arrays

A block $X \in \mathbb{H}$ superscripted with a natural number $n \in \mathbb{N}$ represents a disconnected array of n identical copies of X .

$$X^n = (\mathcal{F}_{\mathbf{ZI}} \mathbf{R}) X^n \quad (8.57)$$

This notation is defined using the constant list operator in [Equation 8.4](#). It is a matter of technical convenience for blocks of the form X^n to be well defined even when $n = 0$. The vacuous case result of \mathbf{ZI} is a block with no inputs or outputs, which is an identity element for the \mathbf{R} combinator up to behavioral equivalence.

8.9.2 Cascades

A step up from an array of identical disconnected blocks is one of arbitrarily varied blocks with each one connected to its neighbor, called a **cascade** hereafter. A simple way of defining a cascade leverages the permutation bus notation while restricting it to identity permutations. Five families of block combinators expressing cascades, with one for each operator depicted in [Figure 8.14](#), are defined as follows.

$$\begin{aligned} \mathbf{D}_n &= \mathcal{F} \lambda(x, y). x \xrightarrow{\downarrow_n} y \\ \mathbf{F}_n &= \mathcal{F} \lambda(x, y). x \xrightarrow{\leftarrow_n} y \\ \mathbf{C}_n &= \mathcal{F} \lambda(x, y). x \xrightarrow{\rightarrow_n} y \\ \mathbf{L}_n &= \mathcal{F} \lambda(x, y). x \xrightarrow{\leftarrow_n} y \\ \mathbf{U}_n &= \mathcal{F} \lambda(x, y). x \xrightarrow{\nearrow_n} y \end{aligned}$$

The names are mnemonic respectively for “down”, “first”, “center”, “last”, and “up”. Each combinator takes a non-empty list of blocks to a single block consisting of each item in the list connected to the next by a bus of width n . For example, to express a cascade of four blocks with each connected centrally to the next by an eight line bus, we could write

$$\mathbf{C}_8 \langle w, x, y, z \rangle.$$

For some choices of bus widths and arities, the associativity might make a difference. If so, right associative evaluation is implied by definition of the folding operator in [Section 8.1.3](#).

$$\mathbf{C}_8 \langle w, x, y, z \rangle = w \xrightarrow{\leftarrow_8} (x \xrightarrow{\leftarrow_8} (y \xrightarrow{\leftarrow_8} z))$$

A bus width value of 1 is implicit in this notation if the subscript n is omitted.

Blockbusters

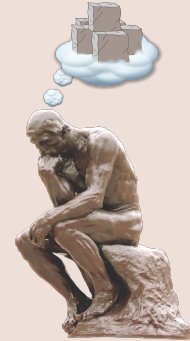
1. To what old song does ρ^3 pertain?
2. Write a program to implement the permutation network combinator definition in [Equation 8.41](#) and confirm that it reproduces [Table 8.1](#).
 - a) What is its asymptotic running time?
 - b) Does this algorithm always produce the shortest valid expression possible (in terms of \mathbf{R} , \mathbf{Z} and \mathbf{I}) for a given permutation network? (hint: Try an exhaustive search.)
3. What general form would the final term have in [Equation 8.54](#)?
4. Is there anything fishy about the function \hat{N} defined by [Equation 8.46](#) reversing the order of the connections?
5. What makes [Equation 8.36](#) an inefficient way to defined the permutation network combinator? (hint: question 2. a) Can it be fixed?
6. Let a block $X \in \mathbb{H}$ have input arity I_X and output arity O_X . Consider the behavioral equivalence

$$(X \uparrow O_X) \stackrel{\alpha}{\cong} (X \downarrow O_X) \stackrel{\alpha}{\cong} (X \uparrow I_X) \stackrel{\alpha}{\cong} (X \downarrow I_X) \stackrel{\alpha}{\cong} X$$

where α is an arbitrary alphabet ordering.

- a) What does this statement mean and why should it be true?
 - b) What definitions would have to be tweaked, and in what way, to make this statement hold not just as a behavioral equivalence but as an equality?
7. Further to [Section 8.3.2](#), a lesser known school of *avant garde* mathematicians (the Boreblocky group) defines a **blockoid** as an abstract algebra (b, r, z, i) consisting of a set b , a binary operator $r : b \times b \rightarrow b$, a unary operator $z : b \rightarrow b$, and a distinguished element $i \in b$. Reverse engineer the algebraic laws that admit only the following structures and natural transformations thereof as blockoid models.
 - $(\mathbb{H}, \mathbf{R}, \mathbf{Z}, \mathbf{I})$
 - $(\mathbb{B}, \mathbf{R}_{\mathbb{B}}, \mathbf{Z}_{\mathbb{B}}, \mathbf{I})$
 - $(\mathbb{L}, \mathbf{R}_{\mathbb{L}}, \mathbf{Z}_{\mathbb{L}}, \mathcal{T}_{\mathbb{L}} \mathbf{I})$

It is acceptable to restrict attention to well formed members of \mathbb{H} , \mathbb{B} , and \mathbb{L} , and to assume a solution to question 6. b. (hint: The equivalence in question 6 should be either axiomatic or derivable from the algebraic laws.)



Part III

Module Families

Great things are not done by
impulse, but by a series of small
things brought together.

Vincent van Gogh

CHAPTER 

AS PRIMITIVE AS CAN BE

A warm welcome is extended to readers who have skipped Part I and Part II to jump directly to the fun part, which begins here. At this point, we are in a credible position to go about establishing a set of primitive components suitable for DI circuit design, which is done in [Section 9.3](#). Also discussed in this chapter are straightforward generalizations of some of the basic primitives to any number of inputs or outputs via tree-like structures, making good use of the block combinators and related notation proposed in [Chapter 8](#).

In the broader context, this chapter inaugurates the development of several families of circuits needed for a general purpose state-based circuit synthesis algorithm in [Chapter 15](#) (that is, one for circuits whose state space is feasible to enumerate). This subproject corresponds to the transformation from the transducer model to the flat netlist representation depicted in [Figure 3.1](#).



Before all that, some unfinished business regarding Petri net and block optimization needs attention. As noted in [Section 8.6](#), transforming a hierarchical block to a primitive block and then optimizing it can facilitate simulation and verification of bottom-up designs. Details about a block optimization method are deferred until [Section 9.2](#) because they depend on the Petri net optimizations discussed first in [Section 9.1](#). The latter are of immediate interest for the current chapter because they simplify the Petri net models of the proposed primitive components from their specifications by process combinators ([Chapter 5](#)) to an obviously correct cruft-free form.

9.1 Petri net optimizations

Along with making Petri nets easier to understand, transformations that simplify them can often make their analysis more efficient. Each place eliminated from a Petri net could reduce the size of its reachability graph by up to half, so it is worthwhile to eliminate any where possible without altering

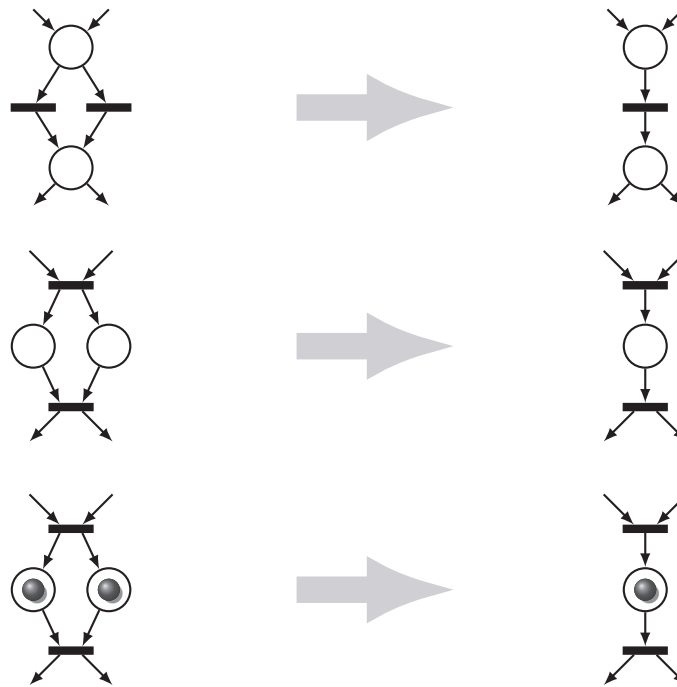


Figure 9.1: A pair of anonymous vertices in a Petri net sharing a common preset and postset may be combined by parallel fusion, provided they are both marked or both unmarked if they are places.

the observable behavior. The set of transformations presented here adapts common features to the current formalism from methods honed by various authors over the years in the areas of asynchronous circuit design [219], software reliability [77, 253] and industrial work flow management [158, 305, 306]. These transformations can be classified as parallel fusion, serial fusion, and cycle removal. The details are discussed in the remainder of this section.

9.1.1 Parallel fusion

The idea of parallel fusion is that any group of Petri net vertices sharing a common preset and postset usually can be combined into a single vertex without changing anything observable. While a properly formal justification of this claim can be found in the cited references, some examples of parallel fusion illustrated in Figure 9.1 may help to justify it intuitively.

- When two transitions are in parallel as shown at the top of the figure, they can only be enabled together. When enabled, only one of them can fire. The choice is immaterial to the subsequent marking, so there is no need for two of them.
- When two unmarked places are in parallel as shown in the middle of the figure, they can only become marked simultaneously, and until then they inhibit their postset transition, as either could do alone. If they were to become marked, they would enable the same postset transition as they would if there were only one of them.

- When two marked places are in parallel, they indicate that it is unsafe for their preset transition to fire, and that their postset transition is enabled. These same conditions could be expressed by just one marked place. When the postset transition fires, the situation reverts to the unmarked case.

These examples do not cover the case of a marked place in parallel with an unmarked place, or of a labeled externally visible transition (*i.e.*, a member of \mathbb{T}), in parallel with anything. The most conservative and correct approach in these cases is to refrain from transforming them, which is no great loss because they are unlikely to occur in practice.

A specification of parallel fusion is straightforward in terms of the Petri net coalescence operator defined in Equation 5.12, the preset and postset notation defined in Section 5.2.2, the set mapping operator μ defined in Equation 5.1, and the partition operator π defined in Equation 6.6. A function $\chi_0 : \mathbb{P} \rightarrow \mathbb{P}$ mapping a Petri net to its equivalent by parallel fusion is defined as the first of several Petri net optimizations.

$$\chi_0(P, T, A, M, F) = (P, T, A, M, F) / \bigcup (\mu \pi \lambda v. (\bullet v, v \bullet)) \{M, P - M, T - \mathbb{T}\} \quad (9.1)$$

This expression can be unpacked as follows.

- The function $\lambda v. (\bullet v, v \bullet)$ maps a vertex v to the pair of its preset and postset vertices in the context of the given Petri net.
- The function $\pi \lambda v. (\bullet v, v \bullet)$ takes a set of vertices and partitions it into equivalence classes of vertices such that within each class, each member has the same preset as any other and the same postset.
- This function is applied by the μ operator to each of the three sets: the marked places M , the unmarked places $P - M$, and the unobservable transitions $T - \mathbb{T}$.
- The Petri net is coalesced with respect to the union of the partitions of each of these three sets, meaning that each vertex is merged with the minimum member of its equivalence class.

9.1.2 Serial transition fusion

The general idea of serial fusion is that a long chain of alternating places and transitions can be shortened to just a single vertex. Serial fusion has more edge cases to consider than parallel fusion when multiple vertices are involved, so it is simpler to proceed by discussing serial transition fusion first in this section followed by a separate treatment of serial place fusion in Section 9.1.3.

Serial transition fusion involves the deletion of a place and the fusion of its preset and postset transitions with each other. Three examples of serial transition fusion are shown in Figure 9.2. The intuition is that whenever the top transition fires, the place gets marked and the bottom one gets enabled by it, so the place might as well be eliminated and the bottom transition enabled directly by whatever enabled the top one.

This transformation is valid only under certain conditions, each of which makes sense on reflection. For one, the place v to be eliminated must not have any other preset transitions, or else there may be another way for it to get marked and hence another way for the bottom transition to be enabled, contrary to the post-fusion semantics.

$$|\bullet v| = 1$$

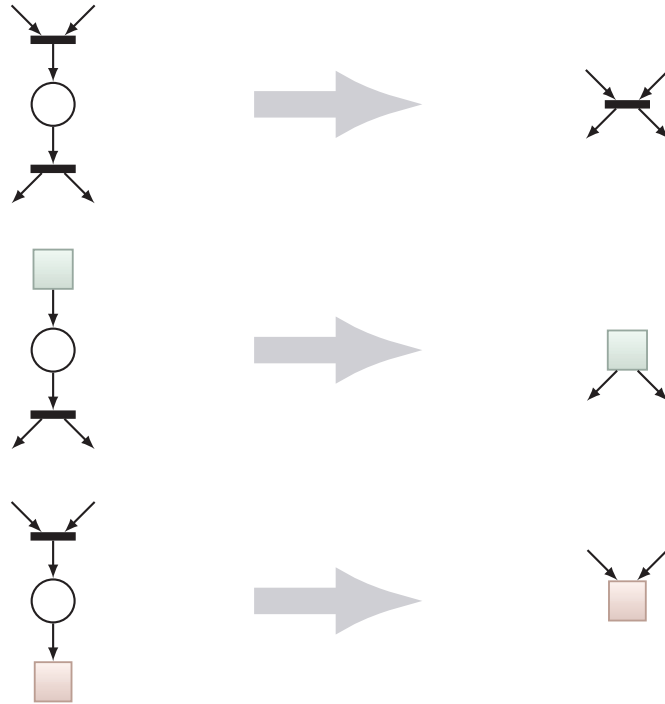


Figure 9.2: An unmarked place with a disjoint unit preset and postset may be deleted and its adjacent transitions merged by serial transition fusion, provided they are not both visible as inputs or outputs.

Furthermore, the place must not have any other postset transitions, or else removing it would eliminate the inhibitory effect on them.

$$|v\bullet| = 1$$

Obviously the preset and postset must also be disjoint, or else this configuration would be a loop rather than a chain.

$$\bullet v \neq v\bullet$$

Before this list of conditions gets any longer, let the set of all vertices in a set S meeting these conditions be abbreviated as follows,

$$\kappa_0(S) = \{v \in S \mid \bullet v \neq v\bullet \wedge |\bullet v| = 1 \wedge |v\bullet| = 1\} \quad (9.2)$$

with the understanding that this notation is meaningful only in the context of a known Petri net inducing the associated preset and postset relations.

Equation 9.2 is relevant to both serial transition fusion and serial place fusion, but there are three further conditions specific to serial transition fusion. First, the place v to be eliminated from a Petri net (P, T, A, M, F) must be unmarked. Otherwise, the postset transition would be already enabled even if the other were not, and this distinction would not be preserved.

$$v \in P - M$$

Second, the preset and postset transitions of v must not both be externally observable, or else there would be no way to fuse them without changing the observable behavior by sacrificing one or the other.

$$(v \bullet \cup \bullet v) - \mathbb{T} \neq \emptyset$$

Third, there must be no other places in the preset of the postset transition. Otherwise, it might not always become enabled under the same conditions as the preset transition, contrary to what their fusion would imply.

$$|\bullet(v \bullet)| = 1$$

A restriction of Equation 9.2 to the set of all places satisfying these three additional conditions for serial transition fusion is easy to express as follows.

$$\dot{\kappa}_1(P, M) = \{v \in \kappa_0(P - M) \mid (v \bullet \cup \bullet v) - \mathbb{T} \neq \emptyset \wedge |\bullet(v \bullet)| = 1\} \quad (9.3)$$

The symmetric restriction $|(\bullet v) \bullet| = 1$ is not required in Equation 9.3 because any additional places in the postset of the preset transition would end up connected to the fused transition pair, which would have the same effect on them. However, fusion in this configuration could sometimes make an open Petri net a closed one. Specifically, if the postset transition is initially an externally visible output, it could acquire a non-empty postset. This effect is not a problem in itself, but it could make any subsequent parallel composition involving the resulting Petri net prohibitively expensive by requiring its prior conversion to canonical form as explained in Section 7.6. A restriction on $\dot{\kappa}_1$ excludes this possibility by allowing a slightly suboptimal Petri net to persist.

$$\ddot{\kappa}_1(P, M) = \{v \in \dot{\kappa}_1(P, M) \mid (v \bullet - \mathbb{V}) \bullet \neq \emptyset \vee |(\bullet v) \bullet| = 1\}$$

In any case, if we take at most one member from the designated set of candidate places

$$\kappa_1(P, M) = \begin{cases} \emptyset & \text{if } \ddot{\kappa}_1(P, M) = \emptyset \\ \{\min \ddot{\kappa}_1(P, M)\} & \text{otherwise} \end{cases}$$

then it is no trouble to write the following specification for serial transition fusion using the componentwise difference operator defined in Section 5.3.3.

$$\chi_1(P, T, A, M, F) = (P, T, A, M, F) / \{\kappa_1(P, M) \bullet \cup \bullet \kappa_1(P, M)\} \dot{-} (\kappa_1(P, M), \emptyset, \emptyset, \emptyset, \emptyset) \quad (9.4)$$

By iterating χ_1 to eliminate one place at a time instead of trying to coalesce whole chains at once, we avoid a technical difficulty, namely the possibility of multiple observable transitions in the same batch.

9.1.3 Serial place fusion

Serial place fusion pertains to a chain of alternating places and transitions similarly to serial transition fusion, but it involves the deletion of an anonymous transition and the fusion of its adjacent places. Some illustrations of this operation are given in Figure 9.3. The idea is that if a single preset place suffices to enable a transition, and the firing of that transition marks only a single postset place, then the transition might as well be removed and the latter place identified with the former.

To be precise, certain conditions are required of any transition suitable for removal. The transition v must have $|\bullet v| = |v \bullet| = 1$, and its preset and postset places must differ. These conditions are

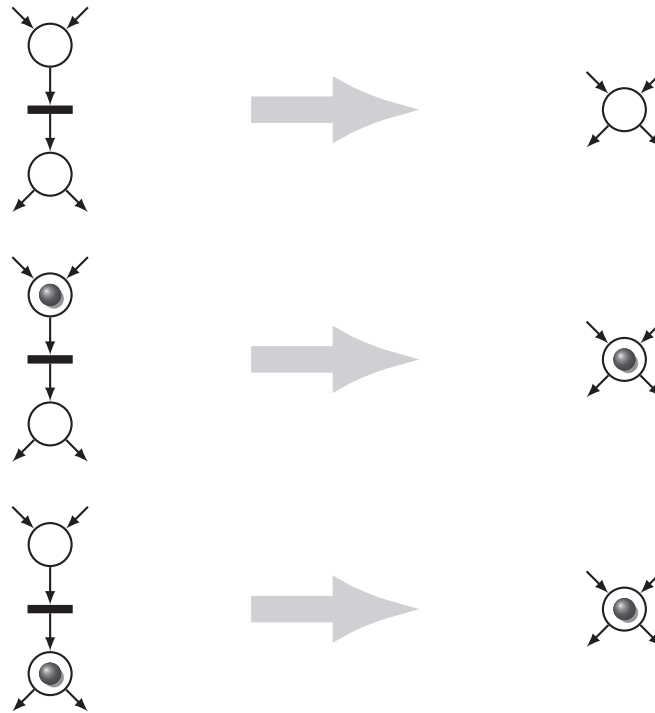


Figure 9.3: An internal transition with a disjoint unit preset and postset may be deleted and its adjacent places merged by serial place fusion, provided they are not both marked.

expressed by Equation 9.2. Three other conditions are also necessary. For a Petri net (P, T, A, M, F) , the transformation requires the transition v not to be externally visible.

$$v \in T - \mathbb{T}$$

Otherwise, its removal could change the trace semantics. It is also necessary for at least one of the preset or postset places of v to be unmarked.

$$(v \bullet \cup \bullet v) - M \neq \emptyset$$

If both places were marked, the Petri net would be unsafe, but fusing the places would hide the issue. Although safety is desirable, changing the semantics arbitrarily is not. For the third condition, the preset place must have no other transitions in its postset.

$$|(\bullet v) \bullet| = 1$$

This condition is necessary because an alternative transition to v would allow another way for the token to flow. It would not be inevitable for the transition v to fire and its postset place to be marked whenever the preset place were marked, so it would not be justified to identify the postset with the preset of v . The set of transitions suitable for removal according to these conditions can be denoted



Figure 9.4: A marked place whose preset coincides with its postset and contains only one transition may be deleted.

as follows (cf. Equation 9.3).

$$\dot{\kappa}_2(T, M) = \{v \in \kappa_0(T - \mathbb{T}) \mid (v \bullet \cup \bullet v) - M \neq \emptyset \wedge |(\bullet v) \bullet| = 1\}$$

Similarly to the case of serial transition fusion, it is simpler to fuse one pair of places at a time than a whole chain. We therefore denote a single member of the set $\dot{\kappa}_2(T, M)$ as follows.

$$\kappa_2(T, M) = \begin{cases} \emptyset & \text{if } \dot{\kappa}_2(T, M) = \emptyset \\ \{\min \dot{\kappa}_2(T, M)\} & \text{otherwise} \end{cases}$$

With that, the definition of serial place fusion comes easily in terms of the coalescence operator (Equation 5.12) and the componentwise difference operator (Section 5.3.3).

$$\chi_2(P, T, A, M, F) = (P, T, A, M, F) / \{\kappa_2(T, M) \bullet \cup \bullet \kappa_2(T, M)\} \dot{-} (\emptyset, \kappa_2(T, M), \emptyset, \emptyset, \emptyset) \quad (9.5)$$

9.1.4 Self-loop place removal

An optimization illustrated in Figure 9.4 does not reduce the number of reachable markings itself, but may be helpful for enabling other optimizations. The optimization is to remove marked self-loop places. A self-loop place is one whose preset and postset contain the same transition. The set of marked self-loop places v in a Petri net (P, T, A, M, F) is captured mainly by

$$\dot{\kappa}_3(M) = \{v \in M \mid \bullet v = v \bullet \wedge |v \bullet| \leq 1 \wedge \bullet(v \bullet) \neq 1\}$$

which also includes disconnected marked places for free. The idea is that if the transition in $v \bullet$ is enabled, then it is enabled anyway without any help from v , and if it is disabled, v does not suffice to enable it. Whether the transition fires or not, the marking of v never changes. Therefore, v has no effect on anything. The requirement of $\bullet(v \bullet) \neq 1$ prevents accidentally creating an open anonymous transition when v is the only member of its postset transition's preset. (See page 91.) If a transition is connected to nothing but self-loop places, not all of them should be removed. Similarly to serial fusion, we therefore take the precaution of removing only one at a time.

$$\kappa_3(M) = \begin{cases} \emptyset & \text{if } \dot{\kappa}_3(M) = \emptyset \\ \{\min \dot{\kappa}_3(M)\} & \text{otherwise} \end{cases}$$

A specification of this optimization by a function $\chi_3 : \mathbb{P} \rightarrow \mathbb{P}$ could hardly be simpler.

$$\chi_3(P, T, A, M, F) = (P, T, A, M, F) \dot{-} (\kappa_3(M), \emptyset, \emptyset, \emptyset, \emptyset)$$

By definition of the componentwise difference operator, removing $\kappa_3(M)$ from the set of places also removes it from the marking and removes any arcs connected to it.

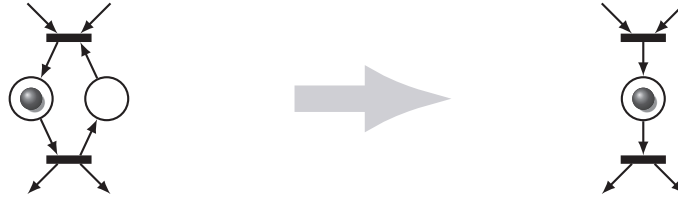


Figure 9.5: An unmarked place that does nothing but regulate the flow of tokens through a marked place can be deleted under certain conditions.

9.1.5 Self-loop transition removal

Removable self-loop transitions are rare in Petri nets derived from process combinator expressions, but can occur frequently in Petri nets derived from recursive or conditional block combinator expressions. By Equation 8.57, any subexpression of the form X^n evaluates to the zero-arity block ZI when n is equal to 0. This block maps to a self-loop place connected to a self-loop transition as in Figure 9.4, but disconnected from the rest of the Petri net containing it. Both the place and the transition can be removed, but not by χ_3 , so a specific transformation for self-loop transition removal is needed. If the associated place becomes isolated as a result, it may be removed by χ_3 or χ_6 (Section 9.1.7) depending on whether it is marked or unmarked.

To specify the optimization formally, a self-loop transition has a place common to both its preset and its postset. For the transition to be suitable for removal, the preset and postset must both have only one member, and the transition must be unobservable. The set of transitions meeting these conditions in a Petri net (P, T, A, M, F) is denoted

$$\kappa_4(T) = \{v \in T - \mathbb{T} \mid \bullet v = v \bullet \wedge |v \bullet| = 1\}.$$

The transformation removing the transitions meeting these criteria is defined as follows.

$$\chi_4(P, T, A, M, F) = (P, T, A, M, F) \div (\emptyset, \kappa_4(T), \emptyset, \emptyset, \emptyset)$$

9.1.6 Redundant cycle removal

One further cycle removal optimization is a bit less obvious but worthwhile because it is applicable frequently. As shown in Figure 9.5, the optimization pertains to a cycle of two transitions and two places, with one place marked and the other unmarked. Under certain conditions, it is reasonable to remove the unmarked place.

Overview

The intuition underlying this transformation relies on the places not being connected to any other transitions outside of the cycle, so that the rest of the Petri net can interact with the cycle only by way of the transitions in it. At the left of Figure 9.5, before the transformation, the unmarked place inhibits the top transition, whereas afterwards at right, there is nothing to inhibit it, so it could be enabled, albeit unsafely, if its preset were marked. While it may seem that this alteration introduces a danger not present initially, there is actually no practical difference. In either case, it is not safe for the preset of the top transition to be marked.

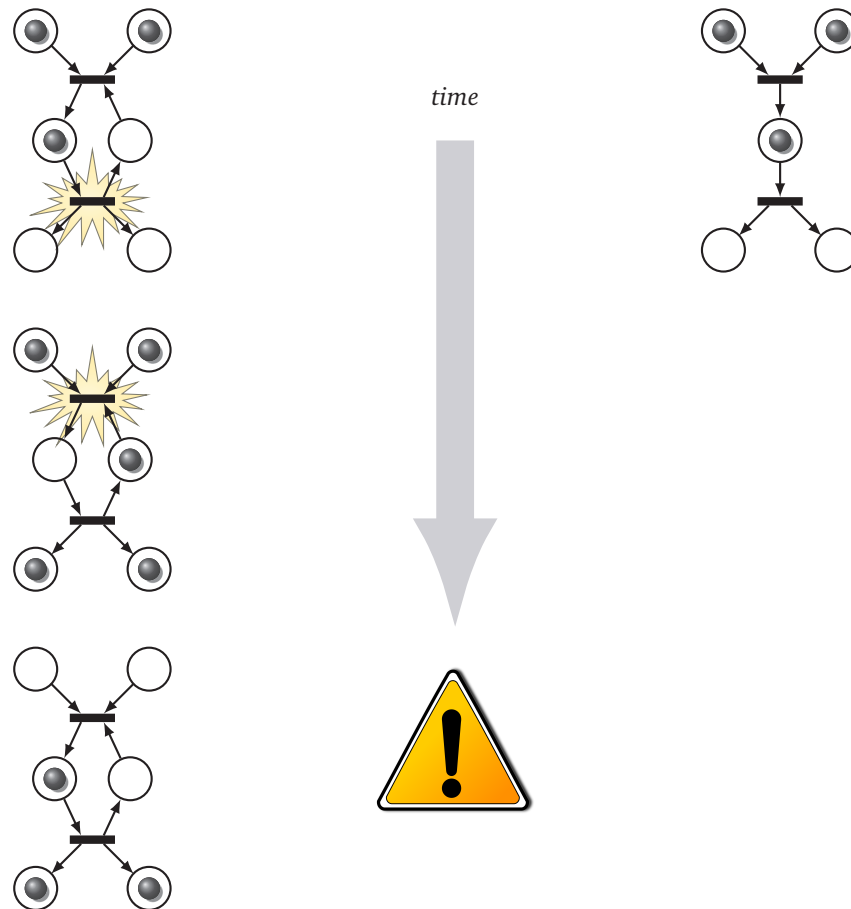


Figure 9.6: If redundant cycle removal can make a Petri net unsafe (upper right), then it would have become unsafe anyway two firings later (lower left).

- With or without the unmarked place in [Figure 9.5](#), the bottom transition is enabled and its firing is imminent. If any arc from the bottom transition leads to a marked place, then the Petri net is already unsafe in either case and hence equivalent with respect to the transformation.
- If arcs from the bottom transition lead only to unmarked places, then they are soon to be marked when the bottom transition fires. This firing also unmarks the marked place and marks the unmarked place in the figure, thereby enabling the top transition if its preset is marked.
- Should the top transition fire, the initially marked place in the figure becomes marked again, and the bottom transition is re-enabled. When the bottom transition fires for the second time, it can overflow the previously unmarked but now marked places in its postset, thereby

violating safety as a direct and inevitable result of the preset of the top transition having been marked. This line of reasoning is illustrated in [Figure 9.6](#).

An attentive reader may notice a loophole in this argument. If the bottom transition had neither any marked nor unmarked places in its postset other than the one shown in [Figure 9.5](#) (i.e., if the outgoing arcs at the bottom went nowhere), then it would be perfectly safe for the preset of the top transition to be marked, because then there would be no places to overflow provided the redundant cycle removal transformation were not performed. The formal specification will have to cover this edge case.

Specification

To start at the beginning, we seek an unmarked place $v \in P - M$ from a Petri net (P, T, A, M, F) as a candidate for redundant cycle removal. Because v should have only a single preset transition and a single postset transition, which differ from each other, we may abbreviate these conditions by writing

$$v \in \kappa_0(P - M)$$

in terms of [Equation 9.2](#). Another condition is that v should be a member of a cycle, which is captured by writing

$$\bullet\bullet v = v\bullet\bullet$$

and that the cycle has one other place, which can be expressed as follows.

$$|v\bullet\bullet| = 1$$

Because the other place is required to be marked, we also have

$$v\bullet\bullet \subseteq M. \tag{9.6}$$

To take care of the loophole mentioned above, the postset of the preset transition of v should contain at least one place other than v .

$$|(\bullet v)\bullet| > 1$$

There is also the requirement that the places in the cycle are connected to no other transitions than those in the cycle, which is not quite implied by these conditions unless we strengthen [Equation 9.6](#).

$$v\bullet\bullet \subseteq \kappa_0(M)$$

The set of all vertices v meeting these conditions is abbreviated

$$\kappa_5(P, M) = \{v \in \kappa_0(P - M) \mid \bullet\bullet v = v\bullet\bullet \wedge v\bullet\bullet \subseteq \kappa_0(M) \wedge |v\bullet\bullet| = 1 \wedge |(\bullet v)\bullet| > 1\}$$

allowing redundant cycle removal to be specified succinctly as a function $\chi_5 : \mathbb{P} \rightarrow \mathbb{P}$ given by

$$\chi_5(P, T, A, M, F) = (P, T, A, M, F) \div (\kappa_5(P, M), \emptyset, \emptyset, \emptyset, \emptyset).$$

9.1.7 Miscellaneous static optimizations

Two further optimizations have no effect on the size of the reachability graph, nor do they enable any other optimizations, but they may simplify a Petri net for purposes of visual inspection and are easy enough to throw in.

Dead code elimination

The first of these optimizations pertains to the removal of regions of a Petri net that can never execute because they are permanently inhibited by an unmarked place with an empty preset. This situation might occur if an infinitely repeating process is mistakenly combined with another process by sequential composition. This optimization is comparable to dead code elimination in conventional compilers [7]. The set of unmarked places with empty presets can be denoted by

$$\kappa_6(P, M) = \{v \in P - M \mid \bullet v = \emptyset\}$$

and the removal of these places and their postsets defined as follows.

$$\chi_6(P, T, A, M, F) = (P, T, A, M, F) \div (\kappa_6(P, M), \kappa_6(P, M) \bullet, \emptyset, \emptyset, \emptyset)$$

Even the observable transitions in $\kappa_6(P, M) \bullet$ may be removed without changing the trace semantics because none of them can ever fire. In general, it may be more effective to iterate this transformation than to apply it just once.

Error place fusion

The remaining optimization pertains to marked places with empty postsets. These typically appear when a closed Petri net representing a process in a restricted environment is converted to its equivalent canonical form (Section 7.5). The passage of control to a so called error place is a deliberate safety violation intended to model divergence of the process. Multiple error places are possible in Petri nets obtained by parallel composition, and they may be fused with impunity. The set of marked places with empty postsets is straightforward to express,

$$\kappa_7(M) = \{v \in M \mid v \bullet = \emptyset\}$$

as is their fusion using the notation defined in Equation 5.12.

$$\chi_7(P, T, A, M, F) = (P, T, A, M, F) / \{\kappa_7(M)\}$$

9.1.8 The whole mix

More sophisticated Petri net optimizing transformations are always conceivable, but the ones specified by χ_0 through χ_7 get rid of most of the cruft that accrues to Petri nets built by the process combinators introduced in Chapter 5, so they will suffice for now.

It should also be noted that although optimization does not alter the trace semantics of a Petri net model, it may violate some of the invariants on which process combinators depend. For example, nothing prevents the removal of every initially marked place from a Petri net if they are all self-loop places, making any subsequent combination by sequential composition difficult. As discussed presently in Section 9.2, these optimizations are more suitable for Petri nets associated with physical components in block diagrams or netlists (to which process combinators other than parallel composition are not relevant), than they are for process combinator operands. Fortunately, parallel composition is not impeded by any of these optimizations.

There are two remaining issues of interest on this subject. As noted in connection with self-loop place removal, it is often the case that one optimization is a prerequisite for another. Also, the discussions of serial fusion and self-loop place mention a need to iterate them exhaustively for the

full effect. A general purpose Petri net optimization function $\chi_{\mathbb{P}} : \mathbb{P} \rightarrow \mathbb{P}$ addresses both of these concerns by iterating the optimizations exhaustively in every useful order.

$$\chi_{\mathbb{P}} = (\chi_7 \circ \chi_6 \circ \chi_5 \circ \chi_4 \circ \chi_3 \circ \chi_2 \circ \chi_1 \circ \chi_0)^\infty \quad (9.7)$$

This function incorporates all of the optimization functions defined hitherto with the limit operator defined in Equation 6.3, and is the function to be used for Petri net optimization in the next section.

9.2 Block optimizations

Part of the motivation for the transformation $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ developed in Section 8.6, which converts a hierarchical block $h \in \mathbb{H}$ to an equivalent primitive block $\mathcal{T}_{\mathbb{H}\mathbb{B}}(h) \in \mathbb{B}$, is more efficient verification when $\mathcal{T}_{\mathbb{H}\mathbb{B}}(h)$ is substituted for h . In some scenarios, efficiency of verification can make the difference between whether or not it is feasible to establish the correctness of a design before production, which is a matter of quite practical interest. However, the block representation $\mathcal{T}_{\mathbb{H}\mathbb{B}}(h)$, amounting as it does to the parallel composition of possibly a large number of Petri net models, is not especially conducive to more efficient verification than the original representation h would be unless it can be simplified in some way. It is now appropriate to revisit this problem with the benefit of the Petri net optimizations developed in the previous section.

9.2.1 Overview

An obvious approach to transforming a hierarchical block $h \in \mathbb{H}$ not just to an equivalent primitive block in \mathbb{B} but to an optimized equivalent is to transform it to an equivalent member of \mathbb{B} first with $\mathcal{T}_{\mathbb{H}\mathbb{B}}$ and then to optimize the result with Petri net optimizations. Unfortunately, a member of \mathbb{B} is not a Petri net, but a triple (I, O, B) with $I, O \in \mathbb{N}$ and $B : \mathbb{T}^I \times \mathbb{T}^O \rightarrow \mathbb{D}$ for good reasons explained in Section 8.2. The Petri net model associated with a block comes only as a result of applying its semantic function B to a specific pair of lists of input and output terminals, and even then only as the Petri net $N \in \mathbb{P}$ within a triple $(I', O', N) \in \mathbb{D}$. To make the semantic function optimize the Petri net after generating it, we could replace a block (I, O, B) with $(I, O, (\lambda(I', O', N). (I', O', \chi_{\mathbb{P}} N)) \circ B)$ using the optimization function $\chi_{\mathbb{P}}$ defined in Equation 9.7, but this solution is not completely satisfactory.



- Although the resulting Petri net is simpler, the semantic function becomes more costly by incorporating the optimization function $\chi_{\mathbb{P}}$. The complexity of the semantic function is already at least proportional to the size of the hierarchical block h from which this primitive block is derived by $\mathcal{T}_{\mathbb{H}\mathbb{B}}$, which is bad enough as it stands.
- This cost multiplies needlessly if the block is used more than once in a design, because substantially similar optimizations are performed for each instance.

A better solution is the seemingly indirect route of transforming the block to a process in \mathbb{D} , optimizing the Petri net in the result just once, and then reverse engineering a block to have a simple semantic function that generates the optimized Petri net immediately. The transformation of a block $X \in \mathbb{B}$ to a member of \mathbb{D} is the easy part, because it is expressible as $\mathcal{T}_{\mathbb{B}\mathbb{D}}(X)$ according to Equation 8.30. Putting it back to a block after optimization would be just as easy by an inverse transformation $\mathcal{T}_{\mathbb{D}\mathbb{B}}$, except that there is no $\mathcal{T}_{\mathbb{D}\mathbb{B}}$ defined in Chapter 8. Such a transformation would

involve generalizing a particular Petri net to a function that generates a Petri net of that form for an arbitrary given alphabet. This problem is not well posed unless an alphabet ordering is also stipulated (page 214), or else there is no way of deciding which terminal on the block to associate with each alphabet symbol of the process. The solution actually proposed below involves a transformation $\mathcal{J}_{\mathbb{B}}^{\alpha}$ comparable to $\mathcal{J}_{\mathbb{D}}^{\alpha}$ defined in Equation 8.33, whereby an alphabet ordering α induces a primitive block $\mathcal{J}_{\mathbb{B}}^{\alpha}(X) \in \mathbb{B}$ for any hierarchical block, netlist, or process $X \in \mathbb{H} \cup \mathbb{L} \cup \mathbb{D}$.

9.2.2 Specifications

Much like the proverbial multi-threaded software application, the problem of block optimization has turned into two problems. One is that of a general transformation from \mathbb{D} to \mathbb{B} as noted in Section 9.2.1, and the other that of somehow sandwiching the Petri net optimizations developed in Section 9.1 into the transformation along the way. The remainder of this section discusses the transformation first before turning to the latter problem.

Transformation

Only the transformation from \mathbb{D} to \mathbb{B} afforded by $\mathcal{J}_{\mathbb{B}}^{\alpha}$ is necessary for the present development, but the cases of \mathbb{H} and \mathbb{L} come relatively cheaply as a byproduct. We therefore define the transformation $\mathcal{J}_{\mathbb{B}}^{\alpha}$ as a recurrence in three cases, the simplest being $X \in \mathbb{H}$, for which $\mathcal{J}_{\mathbb{B}}^{\alpha}(X) = \mathcal{J}_{\mathbb{H}\mathbb{B}}(X)$. A netlist argument $X \in \mathbb{L}$ is first converted to a process $\mathcal{J}_{\mathbb{L}\mathbb{D}} X$ as defined by Equation 8.32, and then recursively converted to a block $\mathcal{J}_{\mathbb{B}}^g \mathcal{J}_{\mathbb{L}\mathbb{D}} X$ using the generic alphabet ordering $g = \mathbb{G}^{\circ-1}$. For the last and most important case, a process $X \in \mathbb{D}$ determines an equivalent block $\mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha} X \in \mathbb{B}$, with the transformation $\mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha}$ remaining to be defined.

$$\mathcal{J}_{\mathbb{B}}^{\alpha}(X) = \begin{cases} \mathcal{J}_{\mathbb{H}\mathbb{B}} X & \text{if } X \in \mathbb{H} \\ \mathcal{J}_{\mathbb{B}}^{\mathbb{G}^{\circ-1}} \mathcal{J}_{\mathbb{L}\mathbb{D}} X & \text{if } X \in \mathbb{L} \\ \mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha} X & \text{if } X \in \mathbb{D} \end{cases} \quad (9.8)$$

An equivalent block $\mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha} X$ to a process $X = (I, O, N)$ would be of the form $(|I|, |O|, B)$, which is to say with input and output arities $|I|$ and $|O|$ equal to the cardinalities respectively of the input and output alphabets of X . The semantic function B needs to take a parameter $(i, o) \in \mathbb{T}^{|I|} \times \mathbb{T}^{|O|}$ consisting of a pair of lists of alphabet symbols to a process isomorphic to X but with an input alphabet determined by i and an output alphabet determined by o .

An intermediate step to constructing the block $\mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha} X$ is to construct a generically alphabetized process $\mathcal{J}_{\mathbb{D}\mathbb{D}}^{\alpha} X$ by Equation 8.29. Then we only need a function B that returns a process $B(i, o) \in \mathbb{D}$ just like the process $\mathcal{J}_{\mathbb{D}\mathbb{B}}^{\alpha} X$ except with the n -th generic symbol

$$t = \mathbb{G}^{\circ-1} n$$

rewritten to the n -th term of the formal parameter i when n is less than $|I|$, or to the $(n - |I|)$ -th term of the formal parameter o for $|I| < n < |I| + |O|$, or in other words to

$$(i \parallel o)_n = (i \parallel o)_{(\mathbb{G}^{\circ} t)} \in \mathbb{T}$$

in general throughout the process. Vertices t outside input and output alphabets of $\mathcal{J}_{\mathbb{D}\mathbb{D}}^{\alpha} X$ can be the same in $B(i, o)$ as they are in $\mathcal{J}_{\mathbb{D}\mathbb{D}}^{\alpha} X$. It is clear from Equation 8.29 that the rewritable vertices are restricted to

$$(\mu \mathbb{G}^{\circ-1})(I \cup O)$$

in terms of alphabets I and O of the original process $X = (I, O, N)$, so a function that handles either case is expressible as

$$\lambda t. (\lambda k. \langle t, (i \parallel o)_{(\mathbb{G}^\circ t)} \rangle_k) \delta_{\emptyset}^{\{t\} - (\mu \mathbb{G}^{\circ -1}) (I \cup O)}$$

and the rewritten process follows as

$$(\lambda t. (\lambda k. \langle t, (i \parallel o)_{(\mathbb{G}^\circ t)} \rangle_k) \delta_{\emptyset}^{\{t\} - (\mu \mathbb{G}^{\circ -1}) (I \cup O)}) \diamond \mathcal{J}_{\mathbb{D}\mathbb{D}}^\alpha X$$

based on the notation of [Equation 5.11](#). Treating this expression as a function of i and o leads to the desired semantic function

$$B = \lambda(i, o). (\lambda t. (\lambda k. \langle t, (i \parallel o)_{(\mathbb{G}^\circ t)} \rangle_k) \delta_{\emptyset}^{\{t\} - (\mu \mathbb{G}^{\circ -1}) (I \cup O)}) \diamond \mathcal{J}_{\mathbb{D}\mathbb{D}}^\alpha X$$

required for the transformation from $X \in \mathbb{D}$ to $(|I|, |O|, B) = \mathcal{J}_{\mathbb{D}\mathbb{B}}^\alpha X \in \mathbb{B}$ defined as

$$\mathcal{J}_{\mathbb{D}\mathbb{B}}^\alpha(X) = (\lambda(I, O, N). (|I|, |O|, \lambda(i, o). (\lambda t. (\lambda k. \langle t, (i \parallel o)_{(\mathbb{G}^\circ t)} \rangle_k) \delta_{\emptyset}^{\{t\} - (\mu \mathbb{G}^{\circ -1}) (I \cup O)}) \diamond \mathcal{J}_{\mathbb{D}\mathbb{D}}^\alpha X)) X$$

thereby concluding the specification of $\mathcal{J}_{\mathbb{B}}^\alpha$ in [Equation 9.8](#).

We may note in passing that $\mathcal{J}_{\mathbb{B}}^\alpha$ enables a couple of more ways to discuss refinement. If $X \in \mathbb{D}$ is a process, and $Y \in \mathbb{H} \cup \mathbb{L}$ is either a block diagram or a netlist, then this equivalence holds

$$X \stackrel{\alpha}{\sqsubseteq} Y \iff \mathcal{J}_{\mathbb{B}}^\alpha X \stackrel{\varepsilon}{\sqsubseteq} Y$$

because converting X to a block according to an alphabet ordering α and comparing the block to Y under an empty alphabet ordering is the same as comparing X to Y under α . The empty alphabet ordering is valid in this context because alphabet orderings are irrelevant to comparison between circuits by [Equation 8.33](#), but perhaps it would be more intuitive and less confusing to be able to express a refinement relationship based on an equivalence

$$X \stackrel{\alpha}{\sqsubseteq} Y \iff X \sqsubseteq (\mathcal{J}_{\mathbb{B}}^\alpha)^{-1} Y$$

where we envision the circuit Y being converted to a process according to an alphabet ordering α by an inverse of the transformation $\mathcal{J}_{\mathbb{B}}^\alpha$, and then compared to X with the ordinary refinement relation among processes. If this way of discussing refinement between a circuit and a process seems more straightforward, it may be because it evokes the conventional practice of labeling the terminals on a generic device with meaningful symbols to the specification instead of somehow doing the opposite.

Unfortunately $\mathcal{J}_{\mathbb{B}}^\alpha$ does not have an inverse because it can map multiple circuits or processes to the same result, but we can certainly have a function $\ell : \mathbb{T}^* \rightarrow ((\mathbb{H} \cup \mathbb{L}) \rightarrow \mathbb{D})$ to capture the notion of labeling a block with a given alphabet by defining ℓ as follows.

$$\ell = \lambda\alpha. (\lambda(I, O, B). B(\alpha \upharpoonright I, \alpha \ll I)) \circ \mathcal{J}_{\mathbb{B}}^\alpha \tag{9.9}$$

The function $\ell\alpha : \mathbb{H} \cup \mathbb{L} \rightarrow \mathbb{D}$ is something like an inverse to $\mathcal{J}_{\mathbb{B}}^\alpha$ in that a process X converted to a block $\mathcal{J}_{\mathbb{B}}^\alpha X$ can always be converted back to an equivalent process $X' \equiv (\ell\alpha) \mathcal{J}_{\mathbb{B}}^\alpha X$.

Optimization

To resume the topic of optimization, one further technique should not be overlooked. The nuclear option for a Petri net resistant to the optimizations described in [Section 9.1](#) is to transform its process X to the canonical form $\mathbf{P}(X)$ as detailed in [Section 7.5](#).¹ This transformation may be more effective because it depends only on the externally observable behavior of the process at a global level regardless of how complicated the Petri net model may be internally.



However, unlike the transformations in [Section 9.1](#), this transformation is costly and not necessarily an improvement, especially if the state encoding is chosen suboptimally (page 185). This outcome is to be expected inasmuch as the canonical form is constrained to preserve the invariants required by the process combinators introduced in [Chapter 5](#). To use it nevertheless as an optimization, we are better off thinking in terms of a tentative canonical form

$$\dot{\mathbf{P}}(X) = \begin{cases} \mathbf{P}(X) & \text{if } \|X\| < K \wedge \|\mathbf{P}(X)\| < \|X\| \\ X & \text{otherwise} \end{cases}$$

where $\|X\|$ denotes some freely chosen complexity metric for a process $X \in \mathbb{D}$, for example the number of places in the Petri net model, and K is some freely chosen constant. For example, it is probably futile to attempt to compute canonical forms of Petri nets with many thousands of places, even though Petri nets of that size are well within the realm of feasibility for local optimizations.

If some combination of local and global optimizations can be helpful, there is no need to choose between them. A single expression combining both would be something like this one based on [Equation 9.7](#).

$$\chi_{\mathbb{D}} = ((\lambda(I, O, N). (I, O, \chi_{\mathbb{P}} N)) \circ \dot{\mathbf{P}})^{\infty} \quad (9.10)$$

The exhaustive iteration may be advantageous if the first iteration simplifies the process enough to enable subsequent optimization by $\dot{\mathbf{P}}$ that would not have been feasible initially. This transformation pertains only to processes in \mathbb{D} , but can be generalized to blocks or netlists $X \in \mathbb{H} \cup \mathbb{L}$ by writing

$$((\lambda(I, O, N). (I, O, \chi_{\mathbb{P}} N)) \circ \dot{\mathbf{P}} \circ \mathcal{J}_{\mathbb{D}}^{\epsilon})^{\infty}$$

using the transformation $\mathcal{J}_{\mathbb{D}}^{\alpha}$ defined in [Equation 8.33](#) with an empty list ϵ for the alphabet ordering. However, this result achieves only an optimized process model in \mathbb{D} with a generic alphabet, falling short of the original goal of an optimized primitive block in \mathbb{B} suitable for further use in block diagrams.

The short step from here to the desired solution incorporates the transformation $\mathcal{J}_{\mathbb{B}}^{\alpha}$ defined by [Equation 9.8](#). By substituting $(I, O, \chi_{\mathbb{P}} N)$ above with an expression of the form $\mathcal{J}_{\mathbb{B}}^{\alpha}(I, O, \chi_{\mathbb{P}} N)$, we arrive at a member of \mathbb{B} . The appropriate alphabet ordering α is always $\mathbb{G}^{\circ-1}$ because $\mathcal{J}_{\mathbb{D}}^{\epsilon} X$ has a generic alphabet, so a general purpose block optimization function $\chi : \mathbb{H} \cup \mathbb{L} \rightarrow \mathbb{B}$ can be defined as follows.

$$\chi_{\mathbb{B}} = ((\lambda(I, O, N). \mathcal{J}_{\mathbb{B}}^{\mathbb{G}^{\circ-1}}(I, O, \chi_{\mathbb{P}} N)) \circ \dot{\mathbf{P}} \circ \mathcal{J}_{\mathbb{D}}^{\epsilon})^{\infty} \quad (9.11)$$

That is, $\chi_{\mathbb{B}}$ with the subscript \mathbb{B} refers to the block optimization rather than the Petri net optimization function.

¹not to be confused with the permutation network combinator \mathbf{P} developed in [Section 8.8.2](#)

9.3 DI primitives

The block optimizations developed in the previous section clear the way for the formal specification of a set of primitive components for DI design. The selection of a set of primitives is important because it creates an abstraction boundary between different engineering disciplines. On one side are those conversant with the physics and manufacturing technologies of circuit fabrication suitable for asynchronous design (e.g., [249]), and on the other are readers and writers of books like this one. While they have the serious job of ensuring that each primitive component has the necessary electrical or chemical properties to serve its purpose, our remit is limited to connecting the components together not too ineptly. In this section, a workable set of primitives is proposed and its basic consequences established.

9.3.1 The continuing saga

The choice of a set of primitives is not set in stone, but nor should it change with the wind. Similarly to the case of a stable software API or the instruction set of a conventional computer, retooling likely incurs a cost. This consideration motivates the search for a set of primitives that is resistant to obsolescence or “future-proof” in some sense. Other desiderata are that the primitives should be few in number, simple to describe, low in arity, and of course efficient to implement. The choice of primitives to be advocated presently builds on various historical influences.

- The set of primitive modules developed for the Macromodules project in the 1960s predated and anticipated many current ideas about delay insensitivity [58, 213]. While the modules were chosen mainly to support register-transfer, control flow, and arithmetic operations, there was also a capability for arbitration inherent in the *interlock* unit, and a form of barrier synchronization by way of the *junction* unit.
- Fast forward to the seventies, the first formal treatment of DI primitives is to be found in [136], which notably introduced a concept of universality and the insight that arbitration is not achievable by any combination of JOIN or MERGE modules.
- An otherwise uneventful decade, the eighties saw another step forward for DI primitives with the arrival of Micropipelines [273]. A specialized architectural paradigm emphasizing asynchronous elastic buffered channels, it popularized the TOGGLE as a primitive and furthered the cause of arbitration.
- The story was to resume in the nineties, with [222] building directly on [136]. The set of primitives became more imaginative, intuitive, and versatile, especially regarding generalizations of the JOIN, with several primitives reaching the forms currently to be proposed.
- The above-mentioned primitives partly overlapped with those appearing contemporaneously in the literature of trace theory [82, 244]. However, the latter featured an explicitly non-quiescent two-terminal primitive, namely the IWIRE, whose inclusion eliminated the complications otherwise required in the way of complemented terminals.
- As of the current millennium, any universal set of primitives yet published still contains at least one member with five or more terminals. This issue complicates connectivity, especially in non-traditional settings such as regular arrays or cellular automata, to the point where some authors have considered cutting corners on delay insensitivity as a workaround [4, 5].

Although less immediately relevant to this discussion, it should be mentioned in passing that other selections of DI primitives have also been used successfully. Null Convention Logic relies on *majority gates* and *threshold gates* as DI primitives [86, 261], while *handshake components* have been used for both DI and QDI design in another alternative [17, 226, 285]. A rigorous treatment of reversibility informs a different choice of primitives in [200].

9.3.2 Universality

In addition to its theoretical appeal, the concept of universality noted above may be a safeguard against obsolescence. The attraction is that a universal set of DI primitives would suffice to implement any delay insensitive circuit that may ever be needed in the future. One way of stating this condition formally as a property of a set of primitives $p \subset \mathbb{B}$ would be

$$\forall (I, O, N) \in \mathbb{D}. \exists \alpha \in (I \cup O)^*. \exists n \in \mathbb{L} \cap (\mathbb{N}^* \times \mathbb{N}^* \times p)^*. (I, O, N) \stackrel{\alpha}{\sqsubseteq} n \quad (9.12)$$

using the generalized refinement relation under an alphabet ordering defined in Equation 8.34. The expression (I, O, N) can be any well formed DI process specification whatsoever. The expression $(\mathbb{N}^* \times \mathbb{N}^* \times p)^*$ restricts the netlists $n \in \mathbb{L}$ to blocks in the set p whose universality is asserted, and despite this restriction the refinement relation still holds.

The mind recoils at proving Equation 9.12 *ab initio* because a constructive proof would be at least as complicated as a general purpose circuit synthesis algorithm (Chapter 15) and far less useful, but fortunately there is a passable substitute. If p is a known universal set of primitives, and p' is a proposed set of primitives, then p' is also universal if every member of p can be implemented by some combination of members of p' .

$$\forall b \in p. \exists n \in (\mathbb{N}^* \times \mathbb{N}^* \times p')^*. b \stackrel{e}{\sqsubseteq} n \quad (9.13)$$

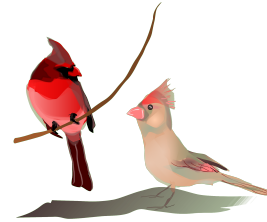
Intuitively we could imagine starting with the netlist n for any given $(I, O, N) \in \mathbb{D}$ containing only blocks $b \in p$ by Equation 9.12, and then flattening each block b into its implementation by p' , resulting in a new netlist n' that implements the original process (I, O, N) but does it using only members of p' . Based on the generally accepted universal set of primitives in [222], proving the universality of the current set by Equation 9.13 entails only the construction of a few specific schematic diagrams and their automated verification by Equation 8.34.

9.3.3 Cardinality and modularity

With this strategy, a universal set of seven primitives is attainable. These can be appropriated mostly from the cited references with technical adjustments to the current formalism, but a few judgment calls are unavoidable on matters of terminology, notation, or philosophy where a consensus appears lacking.

Universal sets of fewer than seven primitives are known, but the cardinality of the set would be an easy metric to game by letting a single unnaturally complex “primitive” emulate others like a programmable gate in an FPGA. A more meaningful figure of merit to minimize is the so called *i/o-modularity* of the set, which is the maximum total arity of any member.

By this criterion, the set to be proposed has an i/o-modularity of 4, making it a long overdue improvement on previously published results and pessimistic conjectures. Lower cardinality with the same or lower i/o-modularity might be desirable but remains an open question. This improvement



is achieved by a minor break with tradition in the way of a novel four-terminal memory or state holding device called a SHUNT, which is not far from a conventional flip-flop element but different enough to require some explanation. Combinations of a SHUNT with other primitives of less or equal arity implement three better known devices of arities 5 and 7, thereby enabling a route to universality by the argument in [222] as planned. Details are given in Section 9.3.5.

9.3.4 Specifications

Most of the seven primitives are specified by an expression of the form $\chi_{\mathbb{B}}(I, O, \lambda t. \text{loop } p(t)) \in \mathbb{B}$, where $p(t)$ is an expression involving the formal parameter t and the process combinators defined in Chapter 5 and Chapter 7, $\chi_{\mathbb{B}} : \mathbb{H} \cup \mathbb{L} \cup \mathbb{B} \rightarrow \mathbb{B}$ is the block optimization function defined by Equation 9.11, and the improvised process combinator $\text{loop} : \mathbb{D} \rightarrow \mathbb{D}$ is defined by Equation 3.5, repeated here for convenience.

$$\text{loop} = \lambda p. \text{fix } \lambda f. \text{seq } (p, f)$$

As explained in Section 3.6.2, this function maps any process p to one that repeats p forever, which is appropriate for the model of a physical component. The block optimization function accounts for the Petri net models having the simplified forms shown in Figure 9.7 and Figure 9.8, and the expression $p(t)$ gives a procedural description of a single loop of the component's operation. The remainder of this section formally defines and briefly describes each primitive individually.

PUSH

The only two-terminal primitive, and the only non-quiescent one, is the PUSH. It sends a signal initially on the output and behaves thereafter as a wire. As the Petri net model in Figure 9.7 shows, it is not safe for the environment to send an input to the PUSH until after the first output signal is transmitted.

$$\text{PUSH} = \chi_{\mathbb{B}}(1, 1, \lambda \langle a \rangle, \langle b \rangle. \text{loop } \text{seq } (\text{put } b, \text{get } a))$$

The schematic symbol for the PUSH is taken from that of the IWIRE device used in trace theory, as is its semantics, but the mnemonic is influenced by related usages in handshake circuits. The change of the mnemonic to a normal word harmonizes it with the other primitive mnemonics, which are usable as both nouns and verbs.

MERGE

The MERGE is a three terminal primitive with two inputs and one output, introduced in advance for motivation in Chapter 2 and Chapter 3 but now formally defined.

$$\text{MERGE} = \chi_{\mathbb{B}}(2, 1, \lambda \langle a, b \rangle, \langle c \rangle. \text{loop } \text{seq } (\text{alt } (\text{get } a, \text{get } b), \text{put } c))$$

As the Petri net model in Figure 9.7 indicates, it accepts an input on either terminal and acknowledges it on the output, but concurrent inputs are not allowed. See Section 3.5.3 for further discussion.

FORK

A FORK is also a three terminal primitive, with one input and two outputs. When it receives a signal on the input, it acknowledges the signal concurrently on both outputs.

$$\text{FORK} = \chi_{\mathbb{B}}(1, 2, \lambda \langle a \rangle, \langle b, c \rangle. \text{loop } \text{seq } (\text{get } a, \text{par } (\text{put } b, \text{put } c)))$$

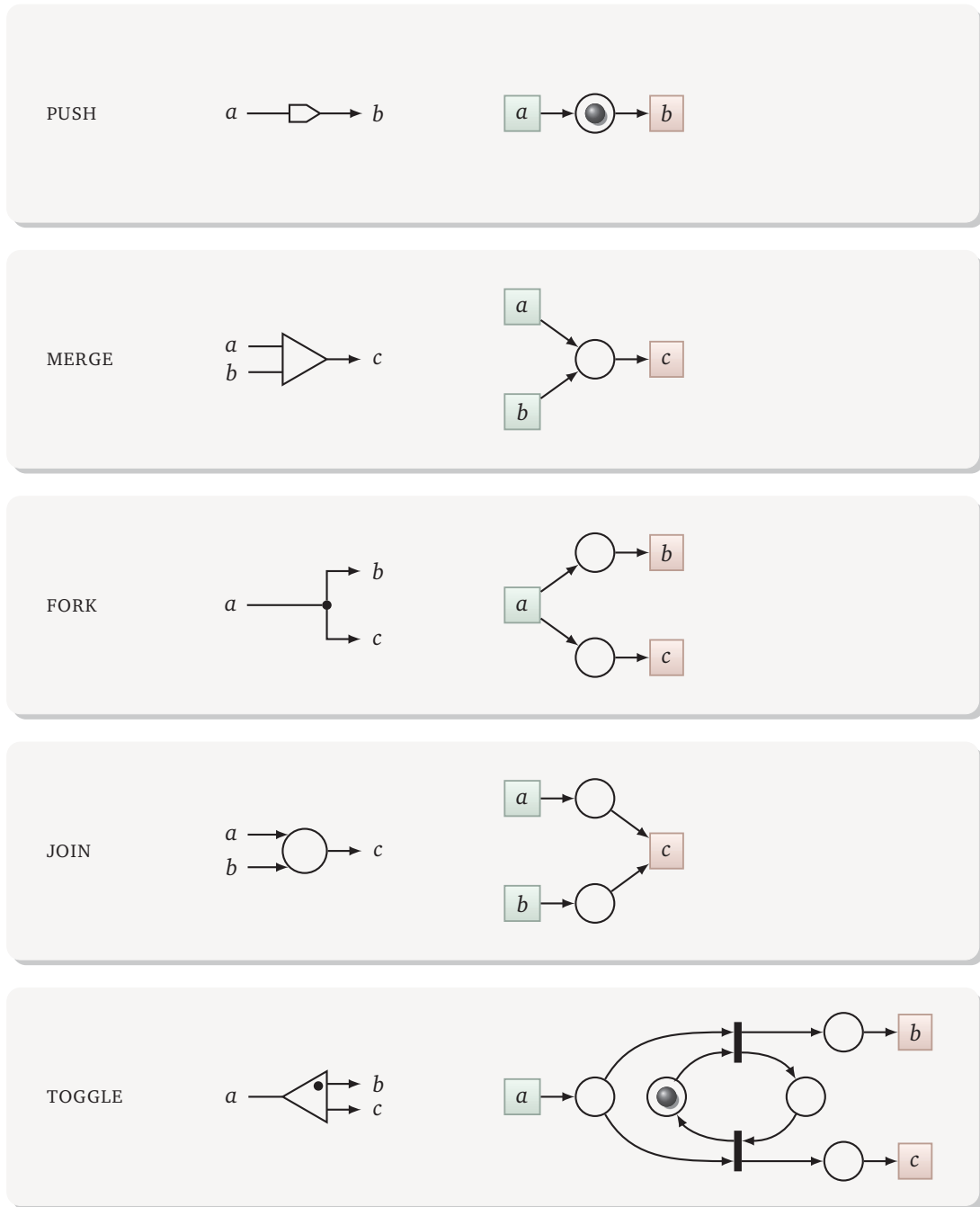


Figure 9.7: two-terminal and three-terminal DI primitive mnemonics, schematic symbols, and Petri net model instances

The protocol implied by the Petri net model in [Figure 9.7](#) constrains the environment to wait until both outputs are observed before sending another input. The output signals may happen concurrently or in either order with unpredictable latency. Contrary to an isochronic fork, detection of either output signal does not imply that the other has been manifested.

Because a FORK is trivial to implement as a split wire in conventional technologies, it hardly seems to merit consideration as a component in itself. However, it is important within the current theoretical framework nevertheless to express the protocol it embodies, and because there is formally no other way to indicate a connection in a circuit from one terminal to two others.

JOIN

The JOIN is another three terminal primitive familiar from some examples in [Chapter 2](#), with two inputs and one output. A pair of concurrent input signals is acknowledged by the output.

$$\text{JOIN} = \chi_{\mathbb{B}}(2, 1, \lambda(\langle a, b \rangle, \langle c \rangle)). \text{loop seq} (\text{par} (\text{get } a, \text{get } b), \text{put } c))$$

A safety condition mentioned informally in [Chapter 2](#) and now evident from the Petri net model in [Figure 9.7](#) is that consecutive signals on the same input are not allowed without an intervening output.

TOGGLE

The TOGGLE primitive, whose behavior is illustrated in [Figure 2.7](#), is the last three-terminal primitive in this list. As noted previously, the TOGGLE has a single input terminal, and input signals are acknowledged alternately by one output terminal or the other.

$$\text{TOGGLE} = \chi_{\mathbb{B}}(1, 2, \lambda(\langle a \rangle, \langle b, c \rangle)). \text{loop} (\text{f seq} \langle \text{get } a, \text{put } b, \text{get } a, \text{put } c \rangle)$$

The specification involves the sequential composition of four events, which is expressed for brevity by folding the `seq` combinator over the list of them using the notation explained in [Section 8.1.3](#).

ARB

Moving on to four-terminal primitives, we have the ARB, which is short for either “arbitrate” or “arbiter” depending on whether the intent is imperative or declarative. This device is also called a *mutex* in some sources, but the ARB mnemonic is preferred here because it is closer to a real word and has the programmer-friendly feature of allowing recognizable truncation of all primitive mnemonics to their first letter.

The purpose of an arbiter is to help the environment manage mutually exclusive access to a shared resource by engaging in only one four-phase (4Φ) handshake at a time.

- There are two input terminals and two output terminals. For the sake of illustration, the inputs are labeled *a* and *c*, and the outputs are labeled *b* and *d*.
- In the absence of contention, a request on *a* is acknowledged on *b*. The next input to *a* signifies a release of the resource, which is also acknowledged on *b*. Similar conventions apply to *c* and *d*.

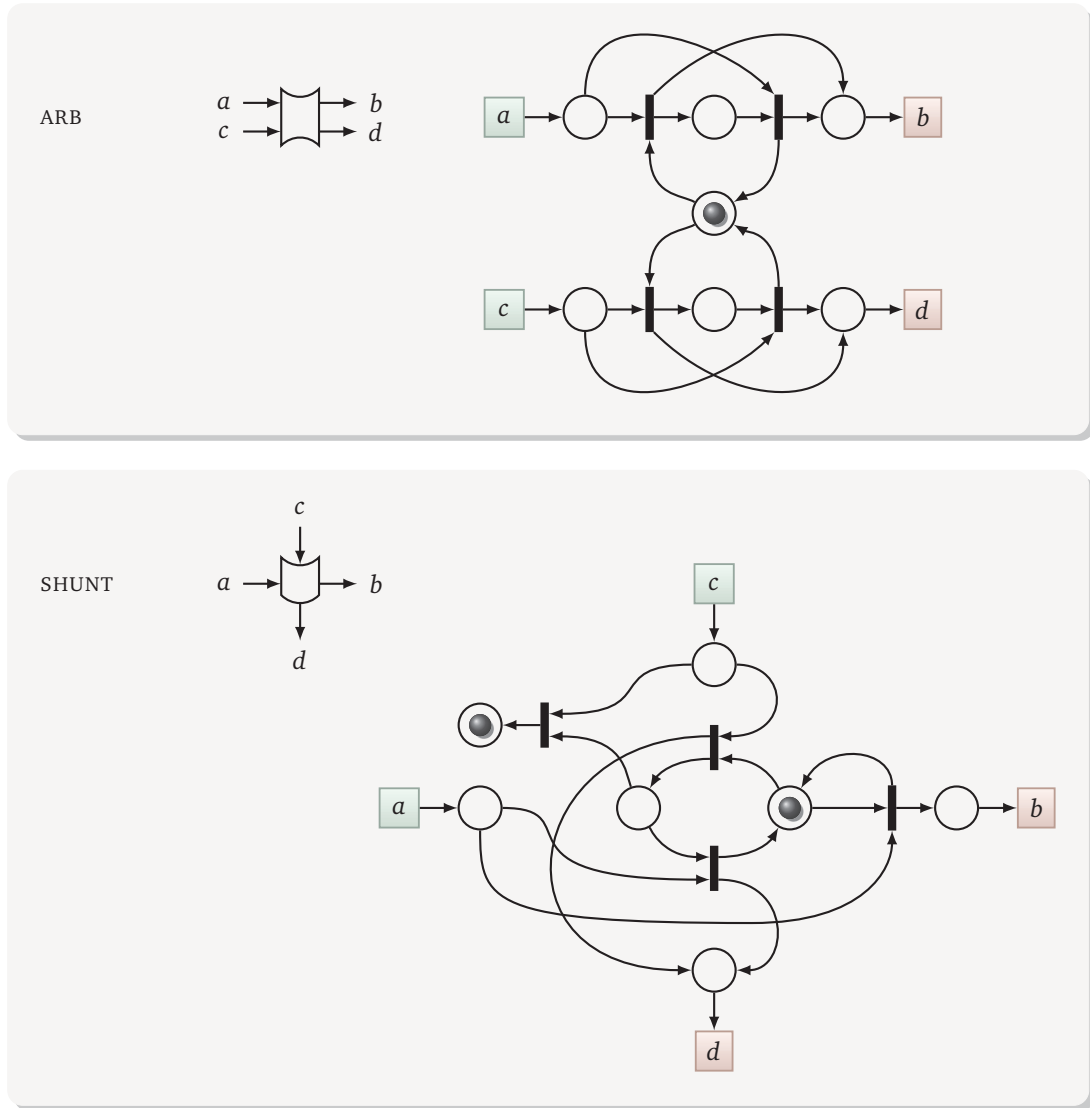


Figure 9.8: four-terminal DI primitive mnemonics, schematic symbols, and Petri net model instances

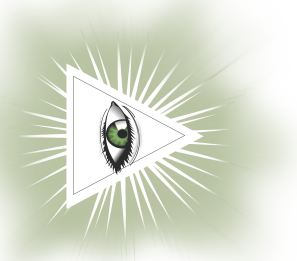
- If a request on c arrives from the environment while the handshake between a and b is in progress, it is buffered until the current handshake completes and the next one can begin. A handshake in progress between c and d similarly precludes the alternative.
- If no handshake is currently in progress and signals arrive simultaneously on both inputs, the ARB makes a non-deterministic choice between them, allowing one 4Φ handshake to complete first and then the other.

The formal specification of an arbiter using process combinators involves a choice between two 4Φ handshakes expressed by folded sequential composition combinators as discussed in connection with the TOGGLE primitive.

$$\begin{aligned} \text{ARB} = (\lambda B. \chi_{\mathbb{B}}(2, 2, B)) \lambda(\langle a, c \rangle, \langle b, d \rangle). \text{loop alt } (& \quad (9.14) \\ & (\text{f seq}) \langle \text{get } a, \text{put } b, \text{get } a, \text{put } b \rangle, \\ & (\text{f seq}) \langle \text{get } c, \text{put } d, \text{get } c, \text{put } d \rangle) \end{aligned}$$

No special provisions for buffering, arbitration, or atomicity are required in this expression, because they are inherent in the **alt** process combinator semantics.

There is no standard schematic symbol for an arbiter. A default choice would be a square box. Some authors have used a box inscribed with “ME” [28, 72, 73, 247, 309] or \mathcal{A} [191], and others have used an unexplained glyph that looks something like \triangleright [181, 294, 295]. The constricted box symbol in Figure 9.8 is used in this book to distinguish the ARB from the proliferation of other boxes in block diagrams. The shape is meant to be a visual reminder that only one handshake can fit through it at a time, and to evoke the general outline of its Petri net model in Figure 9.8, which is obtained automatically after optimization from the process combinator expression in Equation 9.14.



SHUNT

The last primitive in this set is a four-terminal device called a SHUNT. As mentioned previously in Section 9.3.3, it furnishes the missing link to universality by enabling a rudimentary form of mutable memory or state, which no combination of the other primitives in this set can provide. In general, a memory element stores one bit of information, and the environment may set, clear, or read it. Given the limitation to four terminals, a delay insensitive memory element practically designs itself.

- There must be at least two input terminals, or else there would be no way for the environment to communicate its choice of state to the device.
- There must be at least two output terminals, or else there would be no way for the device to distinguish between expressing one state and the other in response to a read request.
- Because there can be no more than four terminals, there must be exactly two input terminals and two output terminals.
- With only two input terminals available, one of them must effect at least two of the three set, clear, or read request operations needed for a memory element.

- We can rule out any protocol that involves setting and clearing the state by the same signal, because these two requests contradict each other.
- A read request signal therefore also must indicate either a set or a clear request atomically, depending on a design decision.
- Whatever the read request does not also indicate (either set or clear) must be requestable separately by a signal to the other input terminal.
- With only two output terminals available, at least one of them must serve the dual purpose of expressing a possible state in response to a read request, and acknowledging a set or clear request.

Having mapped the design space, we have only to choose from a narrow range of protocols consistent with it. Suppose the two input terminals are labeled a and c , and the two outputs are labeled b and d . Let an input signal to a atomically read and clear the current state, and let an input to c set it. In response to a read request on a , an output signal from b indicates a (formerly) clear state, and an output from d the alternative. The remaining design decision is the choice between outputs b and d to acknowledge a set request on c . For reasons to be clarified momentarily, the choice of d is preferable.

The designation of this primitive as a SHUNT rather than a memory derives from an alternative view of its operation: it conditionally routes an input signal to one of two outputs depending on whether it has previously received a control signal. The routable signal is to the input a , and it normally goes to the output b . The control signal is to the input c , and it causes the signal to a to be routed to d instead. Acknowledging the control signal on d makes it convenient often to separate the control acknowledgment from the shunted signal by connecting the output from d to the input of a TOGGLE.

This understanding leads to a simple way of specifying the SHUNT by process combinators as a choice between two alternative handshaking sequences. One sequence is a simple two-phase handshake involving only a and b , while the other is a sequence of input and output signals on c , d , a , and d .

$$\begin{aligned} \text{SHUNT} = & (\lambda B. \chi_{\mathbb{B}}(2, 2, B)) \lambda(\langle a, c \rangle, \langle b, d \rangle). \text{env} (\\ & \text{loop alt (seq (get } a, \text{put } b), (\neg \text{seq}) \langle \text{get } c, \text{put } d, \text{get } a, \text{put } d \rangle), \\ & \text{loop alt (seq (put } a, \text{get } b), (\neg \text{seq}) \langle \text{put } c, \text{get } d, \text{put } a, \text{get } d \rangle)) \end{aligned}$$

Normally a is acknowledged on b , but if there is an input on c , which is always acknowledged on d , the next input on a is also acknowledged on d . The **env** combinator is used here in a way that may seem redundant, but allows for an easier implementation based on the simplifying assumption that no buffering or arbitration are required. The environment sends only one input at a time, and waits for an acknowledgment before sending another one. It also refrains from initiating two consecutive handshakes on c without any on a inbetween. This process combinator expression generates the Petri net model of a SHUNT shown in [Figure 9.8](#) after optimization.

The lack of an established precedent compels an improvised drafting convention. The schematic symbol suggested for a SHUNT is a deformed box as shown in the figure. Its shape is meant to evoke the image of the control input on c deflecting the input on a toward d instead of straight ahead to its usual destination b , not to mention being vaguely reminiscent of the shape of the Petri net.

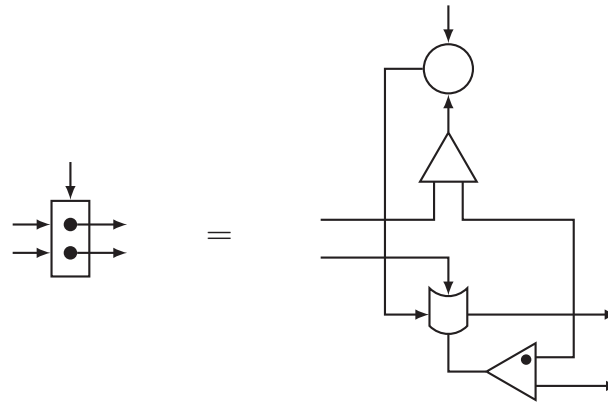


Figure 9.9: the 2-by-1 decision wait schematic symbol and implementation, $DW(2, 1)$

9.3.5 Implications

A set of four primitives with i/o-modularity 7 was argued to be universal in [222] based on [136]. Aside from inconsequential technical differences (e.g., state machines instead of Petri nets), two of them are identical respectively to the MERGE and FORK primitives presented in Section 9.3.4. The third is none other than the 2-way sequencer used as an example in Section 3.6.4 and Section 4.1.1, and the fourth is a multi-way synchronization element not previously discussed. A case for universality of the current set of primitives therefore can be made by exhibiting implementations of the latter two. However, it is convenient to base the sequencer implementation on a similar module called a **decision wait**, which is described first.

Decision wait implementation

An indispensable family of building blocks, decision waits are developed more thoroughly in Chapter 10, but here is a quick preview.

- An n -by- m decision wait has n row inputs, m column inputs, and nm outputs.
- The outputs are depicted as a rectangular array of bullets in n rows and m columns with an implied row-major order.
- Only two input signals are allowed at a time, with one being a row input and the other a column input.
- The decision wait responds to concurrent signals on the i -th row input and the j -th column input with an acknowledgment on the (i, j) -th output.

The complete family of decision waits is given by a function $DW : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking the row and column lengths as natural numbers to a hierarchical block. A full definition of this function follows in Chapter 10, but only a special case is needed presently for the sequencer construction. A schematic for the 2-by-1 decision wait is shown in Figure 9.9 and defined as follows.

$$DW(2, 1) = \langle 0, 2, 1 \rangle \times (\mathbf{ZF}\langle \text{MERGE}, \text{JOIN}, \text{SHUNT}, \text{TOGGLE} \mid 1 \rangle) \quad (9.15)$$

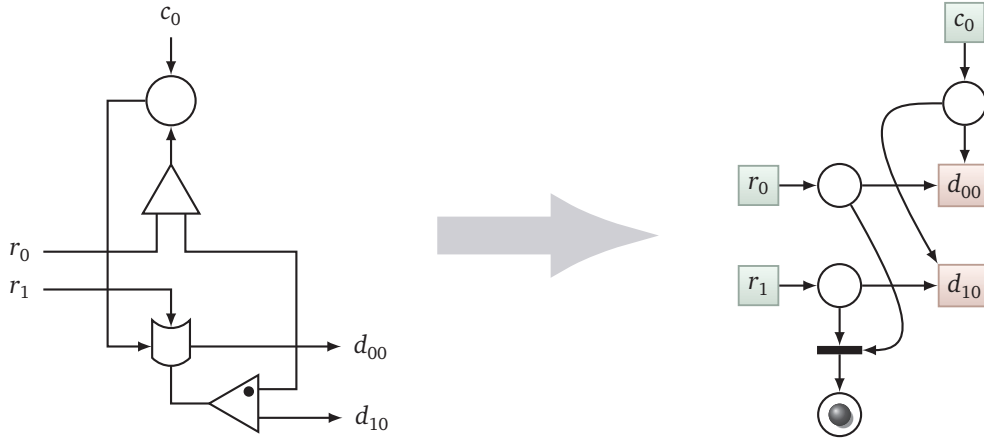


Figure 9.10: Labeling the terminals and transforming a decision wait to a Petri net model by $\ell\langle r_0, r_1, c_0, d_{00}, d_{10} \rangle DW(2, 1)$ confirms correct semantics and the desired terminal ordering.

Using the block combinators defined in Chapter 8, the schematic is captured by a list of the four primitives in the order shown with the first output of each connected to the first input of the next by the F combinator. The outputs of the TOGGLE are rotated and the TOGGLE is connected to the MERGE by the Z combinator. The output terminals are already in row-major order according to this expression, but the inputs need reordering by the permutation shown so as to follow the convention of having the row inputs precede the column inputs.

Figure 9.9 contains the first of several manually designed circuits whose correctness is a matter of interest due to subsequent results depending on it. The circuit's observance of both the terminal ordering convention and the decision wait protocol are required. If the theory developed in Part II is to be of any use for verifying it, now is the time.

Up to this point, we have worked with an informal specification of these requirements, but the latter at least can be made precise and reasonably clear by writing a process combinator expression.

$$\begin{aligned}
 X = \text{env} (& \\
 & \text{loop alt} (\text{seq} (\text{par} (\text{get } r_0, \text{get } c_0), \text{put } d_{00}), \text{seq} (\text{par} (\text{get } r_1, \text{get } c_0), \text{put } d_{10})), \quad (9.16) \\
 & \text{loop alt} (\text{seq} (\text{par} (\text{put } r_0, \text{put } c_0), \text{get } d_{00}), \text{seq} (\text{par} (\text{put } r_1, \text{put } c_0), \text{get } d_{10})))
 \end{aligned}$$

That is, the process X we would like the circuit to implement has row inputs r_0 and r_1 , a column input c_0 , and outputs d_{00} and d_{01} that acknowledge the inputs appropriately. It needs to work only in a benevolent environment devoid of any contention between row inputs.

As a member of \mathbb{H} , the decision wait can be converted to a process in \mathbb{D} for comparison with X by the transformation $\ell\alpha$ defined by Equation 9.9 for some choice of α . An argument to ℓ of $\alpha = \langle r_0, r_1, c_0, d_{00}, d_{10} \rangle$ generates a function $\ell\alpha$ that labels the first terminal of a circuit with r_0 , the second with r_1 , and so on, when transforming it to a process in \mathbb{D} . Hence, if we compute

$$\ell\langle r_0, r_1, c_0, d_{00}, d_{10} \rangle DW(2, 1)$$

The result should refine X as defined in Equation 9.16 if and only if there are no mistakes in the terminal ordering implied by the definition of $DW(2, 1)$ in Equation 9.15. The Petri net model

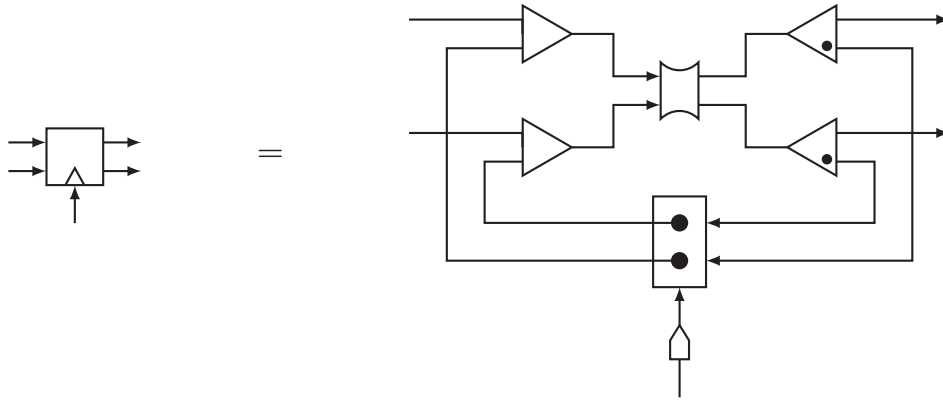


Figure 9.11: A 2-way sequencer is an ARB with a wrapper around it.

obtained by evaluating this expression is shown in [Figure 9.10](#). Some would deem this simple Petri net model obviously correct, but in any case the refinement relationship

$$X \sqsubseteq \ell \langle r_0, r_1, c_0, d_{00}, d_{10} \rangle \text{DW}(2, 1)$$

is computable by [Equation 7.18](#).

Sequencer implementation

The decision wait implementation described above is a stepping stone to that of the sequencer on our current quest for universality, which might be summarized intuitively as an arbiter with a wrapper around it using a decision wait to make it follow a sequencer protocol. As shown in [Figure 9.11](#), the feedback paths via the TOGGLE and the MERGE primitives through the decision wait mean a grant is issued by the sequencer only upon completion of a 4Φ handshake internally by the ARB. The first handshake is allowed to complete because the PUSH initially enables the column input of the decision wait. Subsequent handshakes are blocked by the decision wait until the acknowledgment of the previous grant unblocks them via the PUSH. The decision wait not only synchronizes grants with acknowledgments, but remembers which handshake is in progress.

Generalizations of a sequencer to any number of request and grant pairs are useful enough to merit the construction of a function $\text{SEQ} : \mathbb{N} \rightarrow \mathbb{H}$. This function is fully defined in [Chapter 13](#), but for the present purpose of establishing universality, only the case of $\text{SEQ}(2)$ is needed.

$$\text{SEQ}(2) = \mathbf{Z}^2((\mathbf{Z}^2\mathbf{R}(\mathbf{C}_2 \langle \text{MERGE}^2 \leftarrow_2^1, \text{ARB}, \text{TOGGLE}^2 \rightarrow_2^1 \rangle), \mathbf{D} \langle \text{PUSH}, \text{DW}(2, 1) \rangle)) \Downarrow 2) \quad (9.17)$$

This expression is mainly a straightforward transcription of the schematic in [Figure 9.11](#). A pair of identical blocks in parallel with each other is indicated by squaring the block as explained in [Section 8.9.1](#). A cascade of the MERGE pair, the ARB, and the TOGGLE pair is made by the \mathbf{C}_2 combinator, which connects the first two outputs of each to the first two inputs of the next. Input terminal rotations on the MERGE pair and output terminal rotations on the TOGGLE pair imply a connection to one of each. This cascade is connected to the combined PUSH and decision wait block, with the loops closed by the \mathbf{Z}^2 combinators.

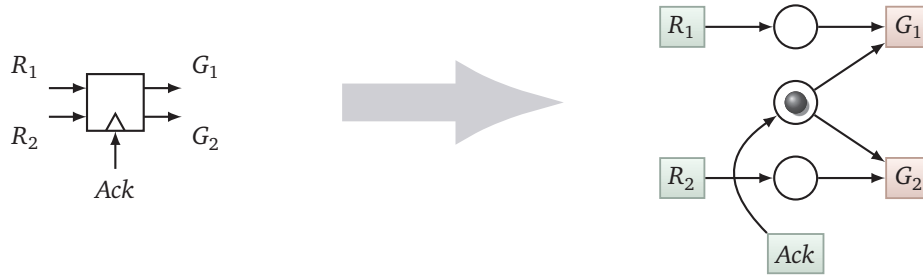


Figure 9.12: Petri net model of $\ell\langle R_1, R_2, Ack, G_1, G_2 \rangle \text{SEQ}(2)$

As usual, the sequencer design and its justification up to this point are pure hand waving and can not be trusted without formal verification. If it is correct, the block defined by Equation 9.17 should associate requests with the first two input terminals, an acknowledgment with one remaining input terminal, and grants with each of two outputs in the same order as the corresponding requests, all following a sequencer protocol. Some confidence is gained by running through the computation of

$$\ell\langle R_1, R_2, Ack, G_1, G_2 \rangle \text{SEQ}(2)$$

according to Equation 9.9 to yield the process whose Petri net model is shown in Figure 9.12, a striking example of block optimization if nothing else. Checking this result by Equation 7.18 for refinement against the sequencer specification in Equation 3.6 can lend it further credibility if its correctness is not sufficiently obvious from the figure.

LJOIN implementation

After the FORK, MERGE, and sequencer, the remaining ingredient for universality according to [222] is a synchronization element called the LJOIN. This device belongs to a useful family of circuits called sparse decision waits. In sparse decision waits, not every combination of row and column inputs is valid. In the simplest case, the LJOIN has two row inputs r_0 and r_1 and two column inputs c_0 and c_1 , but only three outputs, d_{00} , d_{10} , and d_{01} , because the combination of r_1 and c_1 is not acceptable. The schematic symbol is an L-shaped array of bullets similar to that of a decision wait, with an empty space in the position of the excluded output.

The LJOIN has meaningful generalizations to arbitrary row and column lengths, and the full range of possibilities is covered in Chapter 11 but is deferred because only this is needed at present for universality. It can be defined as follows.

$$\text{LJOIN} = (\mathbf{Z}^2(\mathbf{U}\langle \mathbf{L}_2\langle \mathbf{F}\langle \text{TOGGLE, MERGE} \rangle^2, \text{JOIN} \rangle, \mathbf{F}\langle \text{SHUNT}^2 \rangle \uparrow 1 \rangle \parallel 2)) \leftarrow_{\frac{1}{2}} \times \langle 2, 1, 0 \rangle \quad (9.18)$$

As the schematic in Figure 9.13 might suggest, it is an *ad hoc* design based on an embellishment of the decision wait shown in Figure 9.9. The only regular structure is the SHUNT, TOGGLE, and MERGE cascade repeated twice. There are multiple ways this circuit could be expressed, but a good choice is to start by treating the TOGGLE, MERGE, and JOIN subnetwork as a separate unit from the SHUNT cascade as shown here. (See Equation 8.4 and Equation 8.52 for reminders about this notation.)

The formal verification of this circuit is less straightforward than previous examples. A Petri net model derived from the expression $\ell\langle r_0, r_1, c_0, c_1, d_{00}, d_{01}, d_{10} \rangle \text{LJOIN}$ would lack the simplicity

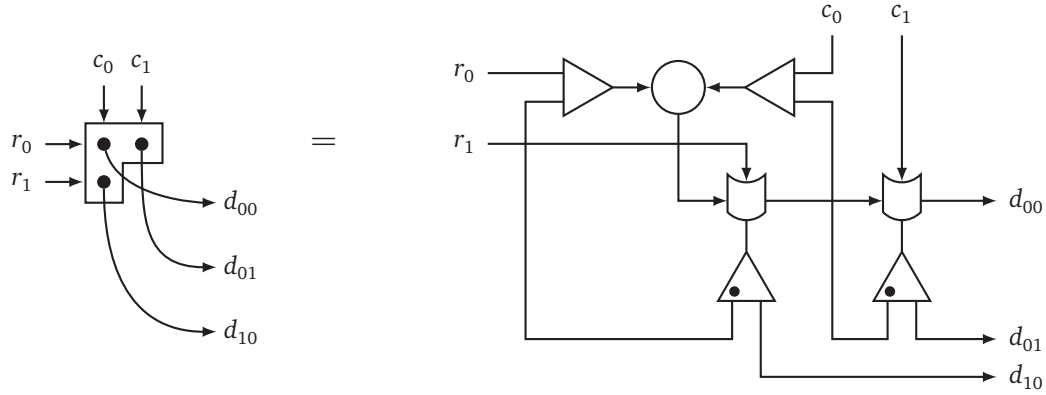


Figure 9.13: The synchronization element LJOIN is an example of a sparse decision wait. Only three combinations of inputs are valid.

and obvious correctness evident in [Figure 9.10](#) and [Figure 9.12](#), because the implementation is not equivalent to the specification. Whereas the effect of concurrent inputs to r_1 and c_1 is undefined as far as the specification is concerned, the circuit accepts them without diverging and issues an acknowledgment on d_{10} . Divergence follows only if the next pair of concurrent inputs includes c_1 , which would entail a second consecutive control input to a SHUNT. The circuit can avoid divergence indefinitely if subsequent inputs are always r_1 and c_0 , or can return to its initial state given an input of r_0 and c_0 (albeit with the wrong acknowledgment). This analysis is confirmed by inspection of the transducer model shown in [Figure 9.14](#), which can be derived automatically from the process model. (See [Equation 7.12](#) for the definition of a transducer.)

Those possessed of a certain delicate intuitive sensibility might deem the circuit in [Figure 9.13](#) compatible with the specification based solely on [Figure 9.14](#) because only the initial state matters. If it had only the initial state, the circuit would be precisely equivalent to the specification, and the other state can be ignored because it is reached only by a prohibited input combination. Whatever the merits of this argument, there is no harm in taking a systematic approach as insurance. A formal specification of the behavior required of the LJOIN is similar to that of the decision wait (cf. [Equation 9.16](#)).

$$\begin{aligned}
 X = \text{env} (& \\
 & \text{loop } (\mathcal{F} \text{ alt}) \text{ seq}^* \langle \\
 & \quad (\text{par } (\text{get } r_0, \text{get } c_0), \text{put } d_{00}), \\
 & \quad (\text{par } (\text{get } r_0, \text{get } c_1), \text{put } d_{01}), \\
 & \quad (\text{par } (\text{get } r_1, \text{get } c_0), \text{put } d_{10}) \rangle, \\
 & \text{loop } (\mathcal{F} \text{ alt}) \text{ seq}^* \langle \\
 & \quad (\text{par } (\text{put } r_0, \text{put } c_0), \text{get } d_{00}), \\
 & \quad (\text{par } (\text{put } r_0, \text{put } c_1), \text{get } d_{01}), \\
 & \quad (\text{par } (\text{put } r_1, \text{put } c_0), \text{get } d_{10}) \rangle)
 \end{aligned}$$

This expression allows for automated confirmation of $X \sqsubseteq \ell \langle r_0, r_1, c_0, c_1, d_{00}, d_{01}, d_{10} \rangle$ LJOIN, thereby

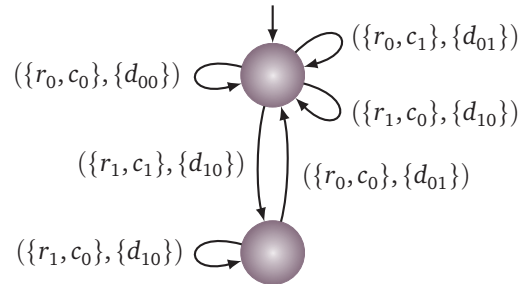


Figure 9.14: The transducer model computed from $T(\ell\langle r_0, r_1, c_0, c_1, d_{00}, d_{01}, d_{10} \rangle \text{LJOIN})$ shows how the circuit refines the specification unnecessarily. Only the initial state is required.

conclusively supporting the claim of universality for the set of primitives presently proposed. This result is reassuring in view of our longer term agenda of developing a general circuit synthesis algorithm to target this set.

9.4 Generalized DI primitives

The primitives developed in [Section 9.3.4](#) are certainly powerful enough to implement any DI process by themselves, but designing a few well chosen modules from them first and then building a circuit by putting the modules together can make the job easier and the results more understandable. Many useful modules are not just one of a kind, but members of a family of modules parameterized by numbers, such as the 2-way sequencer or the 2-by-1 decision wait noted previously in [Section 9.3.5](#), which are instances of the n -way sequencer family or the n -by- m decision wait family respectively. Having defined the initial set of primitives, we turn now to organizing them effectively along these lines.

Before tackling the more sophisticated module families in subsequent chapters, we can practice on some that are both necessary prerequisites for them and conceptually more approachable as straightforward generalizations of the existing primitives to higher arities. Each of the three-terminal primitives generalizes as one might expect to a multi-way version as a tree structure, whereas the ARB primitive generalizes to a multi-way arbiter by sometimes more interesting means. The former are discussed together in [Section 9.4.1](#), and the latter in [Section 9.4.2](#).

9.4.1 Three-terminal primitive generalizations

The protocol followed by a MERGE, JOIN, FORK, or TOGGLE primitive extends intuitively to any number of inputs or outputs. It is easy to imagine a circuit that does what a MERGE would do if it had more than two inputs: accept a signal on any one of its inputs and acknowledge it on the output, with multiple concurrent inputs prohibited. Similarly, a multi-way version of a JOIN would issue an acknowledgment only after a signal were received on every input. A multi-way FORK would acknowledge a single input concurrently on however many outputs it has, and a multi-way TOGGLE would cycle through its outputs sequentially when acknowledging each successive input.

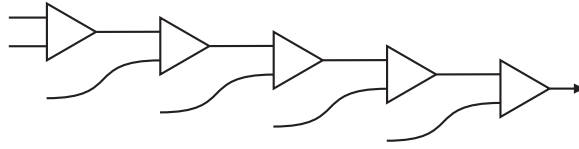


Figure 9.15: A cascaded sequence of five MERGE primitives $F(\text{MERGE}^5)$ implements a 6-way merge protocol.

First attempt

For a MERGE, JOIN or FORK, this behavior is readily achievable by a cascaded sequence as shown in Figure 9.15. For example, the block

$$F(\text{MERGE}^{n-1})$$

would do the job of a MERGE with n inputs for any $n \geq 2$. However, there are two reasons to try harder. One is that this trick fails spectacularly for a TOGGLE. The block $F(\text{TOGGLE}^{n-1})$ generates a complex periodic pattern of output signals with relative frequencies forming a geometric series. The other reason is that the average case *latency* (i.e., the time taken for an input signal to be acknowledged) is proportional to n , the input or output arity, as is its standard deviation assuming uniformly distributed component delays and all input signals equally probable. A logarithmic latency in n would scale better, and it can be achieved at no additional hardware cost.

Logarithmic latency

The way of achieving logarithmic latency is to organize the devices into trees rather than cascades as shown in Figure 9.16. The MERGE and JOIN networks each have eight inputs, but a signal passes through only three devices to reach the output. In general, for n inputs, there would be only $\lceil \log_2(n) \rceil$ devices in the critical path (hence the terminology). Nevertheless, these networks contain the same number of devices as a cascade for a given number of inputs. Similarly, the input to the TOGGLE network passes through only logarithmically many devices to reach the output. The input signal to the FORK network necessarily passes through every node, but delay is still logarithmic because the nodes on each level are traversed concurrently.

Unbalanced trees

Specifying all of the structures in Figure 9.16 would be easy even with that worrisome output permutation in the TOGGLE network if the arity were always a power of two, but in general the trees are not balanced and some sort of workaround is needed. For a multi-way TOGGLE, sometimes the tree is not only unbalanced but technically not even a tree because it must include a feedback path through a MERGE to work correctly. An example of this situation is shown in Figure 9.17. If a 3-way TOGGLE were desired, it would be necessary to build a 4-way TOGGLE first and then feed one of the outputs back through a MERGE to reduce the output arity to 3.

Recurrences

It is possible nevertheless to handle all of these cases with some uniformity. A specification can be developed as a recurrence whereby a tree with arity n is made by combining two trees having an

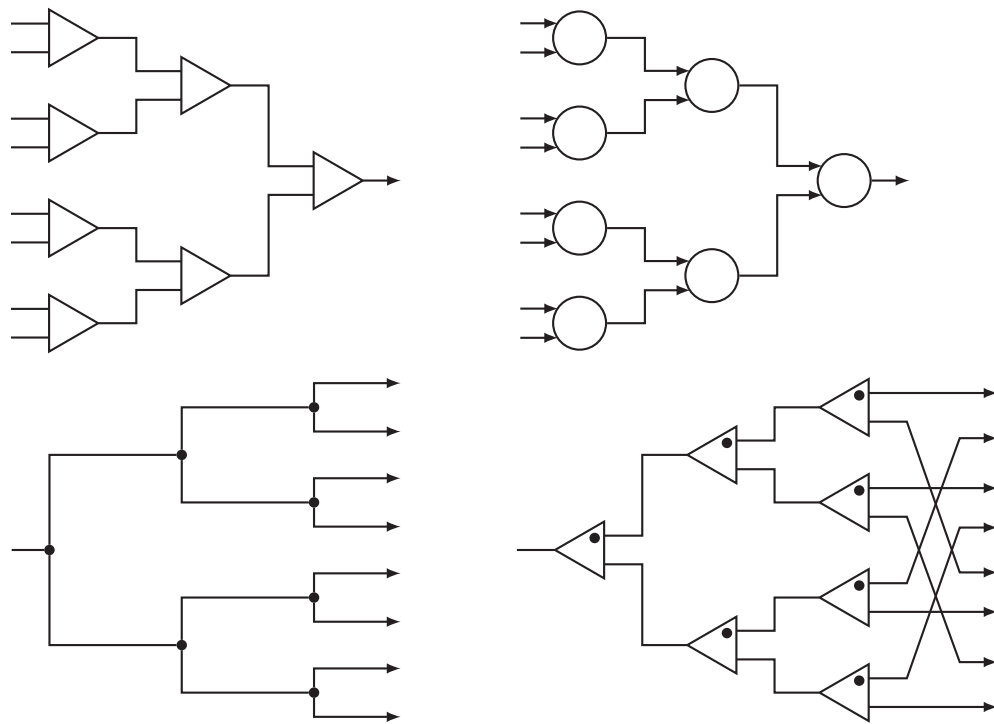


Figure 9.16: Latency-optimized generalizations of three-terminal primitives are made from balanced binary trees where possible.

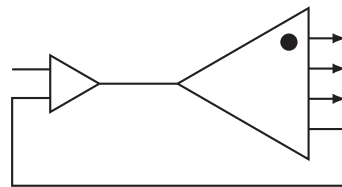


Figure 9.17: A 3-way TOGGLE is made from a 4-way TOGGLE by feeding back one of the outputs through a MERGE.

arity as near as possible to $n/2$ with an additional device as a new root. For a TOGGLE with odd arity, a TOGGLE with the next even arity is created and one of its outputs is fed back through a MERGE to the input. Using camel case typography to distinguish these function mnemonics from their homonymous primitive counterparts, we define each generalized primitive as a function in $\mathbb{N} - \{0\} \rightarrow \mathbb{H}$ with the desired arity n as an argument.

$$\text{FORK}(n) = \langle \mathbf{F}_2 \langle \text{FORK}, (\mathcal{F} \mathbf{R}) \text{FORK}^* \langle [n/2], [n/2] \rangle \rangle, \mathbf{1} \rangle_{\delta_1^n} \quad (9.19)$$

$$\text{MERGE}(n) = \langle \mathbf{F}_2 \langle (\mathcal{F} \mathbf{R}) \text{MERGE}^* \langle [n/2], [n/2] \rangle, \text{MERGE} \rangle, \mathbf{1} \rangle_{\delta_1^n} \quad (9.20)$$

$$\text{JOIN}(n) = \langle \mathbf{F}_2 \langle (\mathcal{F} \mathbf{R}) \text{JOIN}^* \langle [n/2], [n/2] \rangle, \text{JOIN} \rangle, \mathbf{1} \rangle_{\delta_1^n} \quad (9.21)$$

$$\text{TOGGLE}(n) = \langle \mathbf{F}_2 \langle \text{TOGGLE}, (\text{TOGGLE } n/2)^2 \rangle \times \iota_n \times 2, \langle \mathbf{Z}^2 \mathbf{R}(\text{MERGE}, \text{TOGGLE } n + 1), \mathbf{1} \rangle_{\delta_1^n} \rangle_{n \bmod 2}$$

For example, the JOIN function is a member of $\mathbb{N} \rightarrow \mathbb{H}$, whereas the JOIN primitive is a member of \mathbb{B} . It should also be noted that superscripts are used in two different ways. A superscripted function such as \mathbf{Z}^2 denotes \mathbf{Z} composed with itself twice (Equation 6.2), whereas a block with a superscript such as $(\text{TOGGLE } n/2)^2$ denotes a two copies of the block in parallel (Equation 8.57). Furthermore, the output permutation

$$\iota_n \times 2 = \langle 0, n/2, 1, n/2 + 1, 2, n/2 + 2, \dots, n/2 - 1, n - 1 \rangle \quad (9.22)$$

interleaves the TOGGLE outputs as shown in Figure 9.16.

9.4.2 Arbiter generalizations

The remaining primitive having an intuitive generalization to higher arity is the arbiter. A multi-way ARB may have more than two input terminals, along with an output terminal for each. A request (e.g., to acquire a shared resource) is conveyed by the environment as a signal to an input terminal of the arbiter, and each request is acknowledged by the arbiter as a signal from the corresponding output terminal. A subsequent signal by the environment to the same arbiter input terminal (e.g., to release the resource) is acknowledged by the arbiter on the same output terminal, thus completing a 4Φ handshake. Simultaneous requests on any number of inputs are allowed, but only one is acknowledged at a time, and only after any pending 4Φ handshake completes, even if the arbiter has to choose among them non-deterministically. This capability helps the environment manage any number of peers competing for mutually exclusive access to a shared resource up to the number of input and output terminal pairs on the arbiter. It is customary to refer to each associated pair of terminals as a **handshake port**.

Unlike the three-terminal primitives, the ARB has more than one reasonable method of generalizing it to higher arities. For some arities, there is a tradeoff between latency and space efficiency depending on which method is used, and for others the same choice optimizes both. These issues and the importance of arbiters in practice warrant a chapter of their own, so a discussion in depth is deferred to Chapter 12. However, a single example follows in the rest of this section as a taster.

Theory of operation

Designing an n -way arbiter is like organizing a competition wherein each of n contestants is required to compete directly or indirectly against every other in a way that guarantees exactly one winner. It also involves an inductive way of thinking in that we pretend to have solved the problem already for $n - 1$ and need only accommodate one more. One way of putting these two ideas together leads

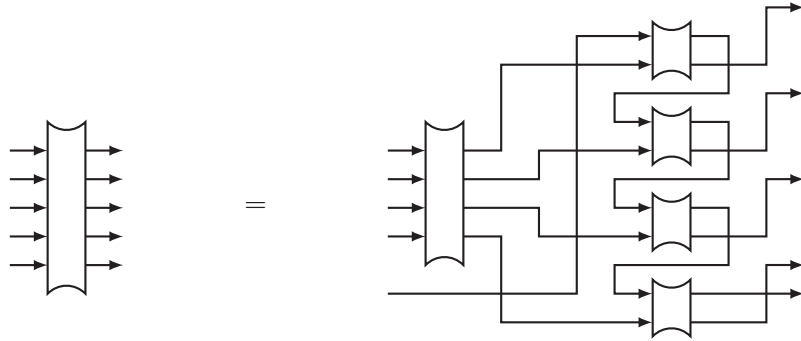


Figure 9.18: An n -way arbiter made from nothing but ARB primitives takes $n^2/2 - n/2$ of them.

to the construction shown in Figure 9.18. The idea of this design is to accommodate a latecomer to a competition among five contestants after the first four have already competed. The overall winner is decided by having the latecomer compete against the winner of the other four, which is done by threading it through a cascade of four ARB primitives so as to meet the 4-way winner wherever it emerges. However, no synchronization is involved. If the latecomer is so late that the winner has already gone through, then we wish it better luck next time as it waits for the winner to complete its handshake, and if it is so early that it whisks through the cascade before any other competitor emerges from the 4-way arbiter, then we congratulate it.

Specification

It is not inordinately difficult express an arbiter of this form through the recursive definition of a function $\text{ARB} : \mathbb{N} \rightarrow \mathbb{H}$.

$$\text{ARB}(n) = (\lambda k. \mathbf{L}_k \langle \text{ARB } k, \mathbf{U}((\text{ARB} \downarrow 1)^k) \rangle) n - 1 \quad (9.23)$$

The cascade of $n - 1$ ARB primitives is given by $\mathbf{U}((\text{ARB} \downarrow 1)^k)$, where the \mathbf{U} operator connects the last output of each term to the first input of the next as explained in Section 8.9.2. Because the ARB outputs are rolled by one, actually the first output of each ARB gets connected to the first input of the next, but the last output from the last ARB becomes the penultimate output from the cascade. In this way, the k outputs of ARB k connected by \mathbf{L}_k to the last k inputs of the cascade correspond to the first k outputs from the whole network, leaving the first input to the cascade and hence the last input to the network disconnected. This input corresponds to the latecomer and is threaded through the cascade to the first input on the last ARB, and hence the last output of the network, as required.

The definition of ARB in Equation 9.23 would be fine except that as a recursive definition it also needs a base case. If $n = 2$, then $\text{ARB}(n)$ should reduce to the ARB primitive, or perhaps $\text{ARB}(1) = \mathbf{I}$ would be more appropriate as a base case. (See Equation 8.15 for the definition of \mathbf{I} .) An elegant solution avoids an overtly recursive definition by reformulating $\text{ARB}(n)$ as a fold over ι_{n-1} using the \mathcal{F} combinator.

$$\text{ARB}(n) = (\mathcal{F}_1 \lambda(i, t). (\lambda k. \mathbf{L}_k \langle t, \mathbf{U}((\text{ARB} \downarrow 1)^k) \rangle) n - i - 1) \iota_{n-1} \quad (9.24)$$

As usual, it would be prudent to verify this design against the desired specification. Viewed as a process, a generalized ARB primitive could have an input alphabet $\{Req_0, Req_1, Req_2, \dots, Req_{n-1}\}$ and

an output alphabet $\{Ack_0, Ack_1, Ack_2, \dots, Ack_{n-1}\}$ when there are n ports, and could be expressed for any given n as $A(n)$ using a function $A : \mathbb{N} \rightarrow \mathbb{D}$ defined as follows.

$$A(n) = \text{loop } (\text{f alt}) (\lambda i. (\text{f seq}) \langle \text{get } Req_i, \text{put } Ack_i, \text{get } Req_i, \text{put } Ack_i \rangle)^* \iota_n$$

If the specification of ARB is correct, then the following refinement relationship should hold. (See [Equation 9.9](#) for the definition of ℓ .)

$$A(n) \sqsubseteq \ell(((\lambda i. Req_i)^* \iota_n) \parallel ((\lambda i. Ack_i)^* \iota_n)) \text{ARB}(n) \quad (9.25)$$

For any fixed n , this proposition can be checked automatically, thereby establishing the correctness of a given candidate circuit for an arbiter with n ports.

Although it is adequate for a practical engineering methodology, [Equation 9.25](#) may be weaker than it looks. Verifying an individual arbiter generated by [Equation 9.24](#) is not equivalent to proving that [Equation 9.24](#) is a correct algorithm for generating arbiters. Correctness of [Equation 9.24](#) would be equivalent to [Equation 9.25](#) universally quantified with respect to n . Higher order propositions like these open the door to undecidability and require *ad hoc* proof techniques. These are not explored further in this book, but we should note in passing that results of this nature could save some of the time normally spent on verification, which can become costly even if it is automated.

The gateless gate

1. In addition to dead code elimination ([Section 9.1.7](#)), what other transformations used in programming language compilers suggest analogous Petri net optimizations, and how might they work? (See [7] for a review.)
2. How could a circuit be designed to implement the LJOIN specification exactly instead of refining it?
3. A process $W = \text{loop seq} (\text{get } a, \text{put } b)$ models a wire, but no WIRE primitive is proposed in [Section 9.3](#) despite its traditional usage in trace theory [82, 244].
 - a) What would be one way of implementing a wire using the given set of primitives?
 - b) What other ways are there? (hint: at least three, excluding trivial variations)
 - c) How are they expressed by block combinators?
 - d) The nullary block combinator $\mathbb{1}$ defined by [Equation 8.15](#) is like a WIRE primitive built into the theory. Is it necessary for universality? (That is, do all of the answers above depend on it?)
4. What is the Petri net model of an isochronic fork? (hint: This question is a kōan.)
5. How long is the critical path of the arbiter depicted in [Figure 9.18](#)? Is it the same for all inputs? If not, what are the worst, best, and average critical paths?



We think in generalities, but we live in detail.

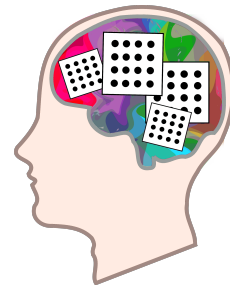
Alfred North Whitehead

CHAPTER 10

DECISIONS, DECISIONS

In this chapter, we resume development of the decision wait family of modules introduced in [Section 9.3.5](#) by generalizing from the 2-by-1 case to n -by- m for arbitrary positive $n, m \in \mathbb{N}$, and then generalize it further to more than two dimensions in preparation for the development of arbitrary dimensional sparse decision waits in [Chapter 11](#).

A sensible reader might well ask why such a small topic warrants two chapters of gnarly schematics and equations and a lengthy appendix featuring more of the same. There are several reasons. The sequencer example incorporating a decision wait discussed previously typifies the practice of requisitioning a decision wait from a mental warehouse as a matter of course, thus avoiding any distraction from the main task at hand. A ready supply of decision waits of all shapes and sizes is a load off the designer's mind because it means any problem that can be reduced to a decision wait can be considered solved. The decision wait is also an advantageous abstraction for automated verification because it has a simple and regular process level description or optimized Petri net model compared to its netlist representation. This property means the protocol it exchanges with its environment is much faster to simulate than whatever takes place internally. Furthermore, the decision wait has a role to play in automated synthesis ([Chapter 15](#)), where it implements a state transition table derived from a transducer model, and in data communication ([Chapter 13](#)), where it makes some types of decoders more efficient.



Even so, could this story not be told more succinctly? The dark side of decision waits is their reputation as large and inefficient structures. A naive formulation would certainly make for a shorter exposition, but would result most likely in circuits with objectionable latencies. Optimal performance demands attention to any factors that may affect it. Similarly to arbiter networks ([Chapter 12](#)), decision waits can be of various general forms, each with considerable scope for further variations within it, and each possibly optimal over different ranges of dimensions, but hard

and fast rules for decision wait decomposition are elusive because there are multiple dimensions to consider and subtle critical path dependences. Instead, we opt for an unbiased approach to enumerating and selectively sampling the design space.

This approach requires consideration of various possible decision wait transformations, such as permutation and rotation, that may affect performance in certain contexts. For example, if an n -by- m decision wait is bigger and slower than an m -by- n decision wait of a similar form, then it makes sense to use the latter in place of the former by permuting the inputs and outputs accordingly. A complete description of such a decision wait would include not only the dimensions n and m , but the classification of its form and an indication of whether or not it has been rotated. If it is composed of lower dimensional decision waits glued together, the same information would be needed in reference to them, and so on.

A formal concept of a decision wait decomposition as a hierarchical structure encapsulating this information turns out to be useful as something like a temporary intermediate representation of a decision wait. It sits between the high level specification $(n, m) \in \mathbb{N} \times \mathbb{N}$, and the low level description $Dw(n, m) \in \mathbb{H}$ to supply all of the details that impact performance but do not interest the designer. The decision wait generating function $Dw : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ is defined in terms of a fixed function \mathcal{U} taking the given dimensions n and m to the optimum decomposition for those dimensions. Such a function, called a **decomposition strategy** in this chapter, would be determined empirically by optimizing over various possibilities with respect to technology dependent factors and updated ideally only as often as the fabrication technology changes.

This chapter is organized as a rabbit hole from which a reader can retreat profitably at several points after some routine math notation is out of the way in [Section 10.1](#). By the end of [Section 10.2](#), everything necessary to the construction of simple two dimensional decision waits of any size is covered. These would be subject to linear latencies, but [Section 10.3](#) takes them a step further to gain logarithmic latency and hence better performance in the limit of large sizes. [Section 10.4](#) introduces two alternative routes to higher dimensional decision waits, one simple and the other more sophisticated, thereby taking the subject as far as possible before involving the rotations and permutations on decision waits proposed in [Section 10.5](#) as a prerequisite to their optimization. Construction of arbitrary dimensional decision waits is revisited in [Section 10.6](#) in terms of general hierarchical decomposition strategies incorporating these transformations. Readers interested in detailed examples of putting these ideas to work at selecting optimum decomposition strategies are referred to [Appendix C](#), which discusses the two metrics of component count and critical path length.

10.1 Ordered trees

A hierarchical structure such as a decision wait decomposition is convenient to package as a **tree**, a term previously used informally but now in need of some firming up. In graph theory, trees are envisioned as acyclic graphs, directed or undirected, with no more than one path connecting any two nodes. The trees needed in this chapter are finite, directed, and connected, with each node having at most one incident edge and any number of outgoing edges. It would be least troublesome philosophically to think of the edges as anonymous but distinguishable and the nodes as labeled, with identical labels possible for nodes that are nevertheless distinct. We stipulate further that among the outgoing edges of any node,



some are deemed to precede others according to a total order relation, hence the terminology of **ordered trees**.

10.1.1 Definition

A simple concrete model is possible for trees meeting these conditions. For a set S , the set $\mathcal{O}(S)$ modeling the ordered trees whose node labels are members of S is the smallest set T satisfying $T = S \times T^*$, or equivalently satisfying the following inductive definition.

- If x is a member of S , then (x, ϵ) is a member of T , where ϵ denotes the empty list.
- If x is a member of S and y is a member of T^* then (x, y) is a member of T .

10.1.2 Terminology

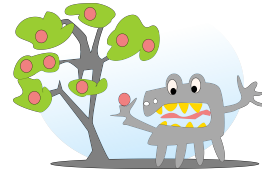
Ignoring the distinction henceforth between the concrete model and the abstract concept of an ordered tree,¹ we regard two ordered trees (x, y) and (x', y') in a set $\mathcal{O}(S)$ as equal to each other if and only if $x = x'$ and $y = y'$, provided equality is defined for members of S . By way of further terminology, for a tree $(x, y) \in \mathcal{O}(S)$

- x is the **root** of the tree
- y is the list of **subtrees**
- and (x, y) is **terminal** if and only if y is ϵ .

For example, if $S = \{u, v, w\}$ is a set, then $(u, \langle (w, \langle (u, \epsilon) \rangle), (v, \epsilon) \rangle)$ is a member of $\mathcal{O}(S)$. It has a root u , a first subtree $(w, \langle (u, \epsilon) \rangle)$, and a second subtree (v, ϵ) , which is terminal.

10.1.3 Computation

Our main use for ordered trees in this chapter is to evaluate decision waits as a function of their decompositions, which are represented as ordered trees. Functions of ordered trees could be expressed as recurrences, but every case that concerns us follows a similar pattern of bottom-up traversal that is simple to encapsulate by a tree folding combinator Λ (Greek upper case lambda) analogous to the list folding combinator f (Section 8.1.3).



The tree folding combinator can be defined as follows. For arbitrary sets S and R , a second order function

$$\Lambda : ((S \times R^*) \rightarrow R) \rightarrow (\mathcal{O}(S) \rightarrow R)$$

takes a function $f : S \times R^* \rightarrow R$ to a function $\Lambda(f) : \mathcal{O}(S) \rightarrow R$ satisfying this recurrence.

$$\Lambda(f) = \lambda(x, y). f(x, (\Lambda f)^* y) \tag{10.1}$$

¹Otherwise to be completely correct we might have to say that equality of models implies no more than a label-preserving and order-preserving isomorphism between the trees they model.

That is, to compute the function $\Lambda(f)$ given a tree $(x, y) \in \mathcal{O}(S)$ as an argument, we would map $\Lambda(f)$ recursively over the list of subtrees y , make a list $v = (\Lambda f)^* y$ of the results, and then apply f to the pair (x, v) .

A function of the form $\Lambda(f)$ always takes a tree as an argument but need not return a tree. For example, a function $h : \mathcal{O}(S) \rightarrow \mathbb{N}$ that maps a tree $t \in \mathcal{O}(S)$ to the number $h(t) \in \mathbb{N}$ of nodes in t could be defined like this.

$$h = \Lambda \lambda(x, v). 1 + \sum_{i=0}^{|v|-1} v_i \quad (10.2)$$

A more interesting example is a function $g : \mathcal{O}(S) \rightarrow S^*$ that takes a tree $t \in \mathcal{O}(S)$ to the list $g(t) \in S^*$ of its node labels in level order

$$g = b \circ \Lambda \lambda(x, v). \langle x \rangle : (\mathcal{F}_\epsilon \lambda(a, b). (\lambda m. ((\lambda i. a_i \parallel b_i)^* \iota_m) \parallel (a \ll m) \parallel (b \ll m)) \min\{|a|, |b|\}) v$$

which would imply $h(t) = |g(t)|$.

10.1.4 Notation

Sums of lists of natural numbers $v \in \mathbb{N}^*$ such as the one in Equation 10.2 appear frequently in this chapter, so a notation omitting the limits

$$\sum v = \sum_{i=0}^{|v|-1} v_i$$

is used hereafter to express the summation over the whole list. The usage

$$\sum(v \upharpoonright n) = \begin{cases} \sum_{i=0}^{n-1} v_i & \text{if } n < |v| \\ \sum v & \text{otherwise} \end{cases}$$

follows from Equation 8.7, with the conventional vacuous sum $\sum(v \upharpoonright 0) = \sum \epsilon = 0$ implicit throughout. The analogous notations $\prod v$ and $\prod(v \upharpoonright i)$ for products also apply, with vacuous products $\prod \epsilon = 1$ per convention. Furthermore, the sum and product operators can also be treated as functions to be mapped over a list of lists $w \in \mathbb{N}^{**}$ in expressions like these

$$\begin{aligned} \sum^* w &= (\lambda v. \sum v)^* w \\ \prod^* w &= (\lambda v. \prod v)^* w \end{aligned}$$

which both reduce to lists in \mathbb{N}^* .

A notion of arrays or lattices also tends to pervade any discussion of decision waits, in the sense of an n -dimensional space of discrete points with each point identified by a list of n coordinates. The idea is readily intuitive that a list of non-zero dimensions $d \in \mathbb{N}^n$ should determine a set of $\prod d$ points $p \in \mathbb{N}^n$ satisfying $0 \leq p_i < d_i$ for all $0 \leq i < n$, but expressing it repeatedly is tiresome unless we take another liberty with notation. A higher dimensional analog \bar{v}_d to the notation ι_n defined in Equation 8.3 pertains to a list $d \in \mathbb{N}^*$ appearing in the subscript, and denotes the list

$$\bar{v}_d = (\mathcal{F}_{\langle \epsilon \rangle} \lambda(h, t). b (\lambda i. (\lambda j. i : j)^* t)^* \iota_h) d \in (\mathbb{N}^{|d|}) \prod^d \quad (10.3)$$

of all points p meeting the condition above ordered lexicographically.

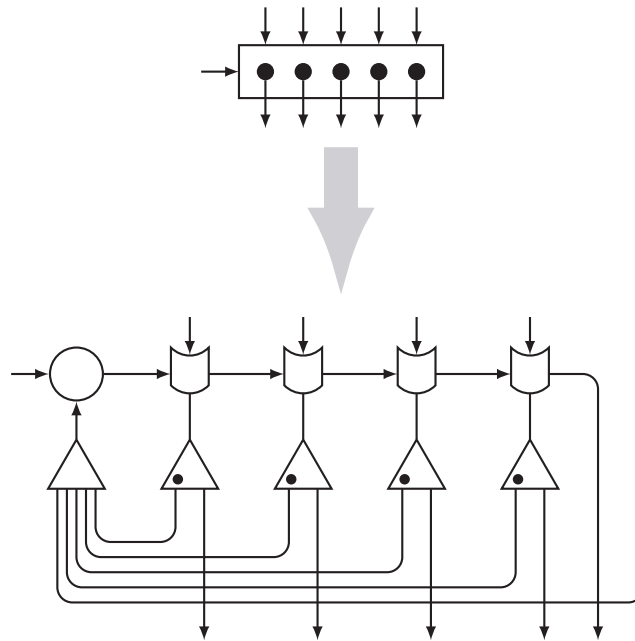


Figure 10.1: A lateral decision wait has one row and any number of columns.

10.2 Cascading planar decision waits

Many useful decision waits have only two dimensions, (*i.e.*, rows and columns), and are called **planar** decision waits hereafter to distinguish them from higher dimensional decision waits. A simple way to construct planar decision waits is by repetitive arrays of standard cells, which are the focus of this section. Decision waits of this form are described as **cascading** for the sake of discussion, as opposed other organizations considered in [Section 10.3](#).

Cascading planar decision waits are simple because building them to arbitrarily large dimensions is only matter of combining sufficiently many cells according to an obvious pattern. By some metrics, (*e.g.*, component count) cascading planar decision waits are also likely to be the best alternative, but perhaps not the fastest for reasons illustrated shortly. Nevertheless, they are the foundation for all subsequent constructions in this chapter.

To make them even simpler, we break down cascading planar decision waits further into three cases. [Section 10.2.1](#) dispenses quickly with **lateral** decision waits, which are those having one row and any number of columns. [Section 10.2.2](#) develops **bilateral** decision waits, which have two rows and any number of columns, and [Section 10.2.3](#) treats the case of sizes greater than two in both dimensions.

10.2.1 Lateral

A minor rearrangement of [Figure 9.9](#) in [Figure 10.1](#) suggests a way of building lateral decision waits by the yard: we need only cascade a SHUNT and a TOGGLE combination onto the right for

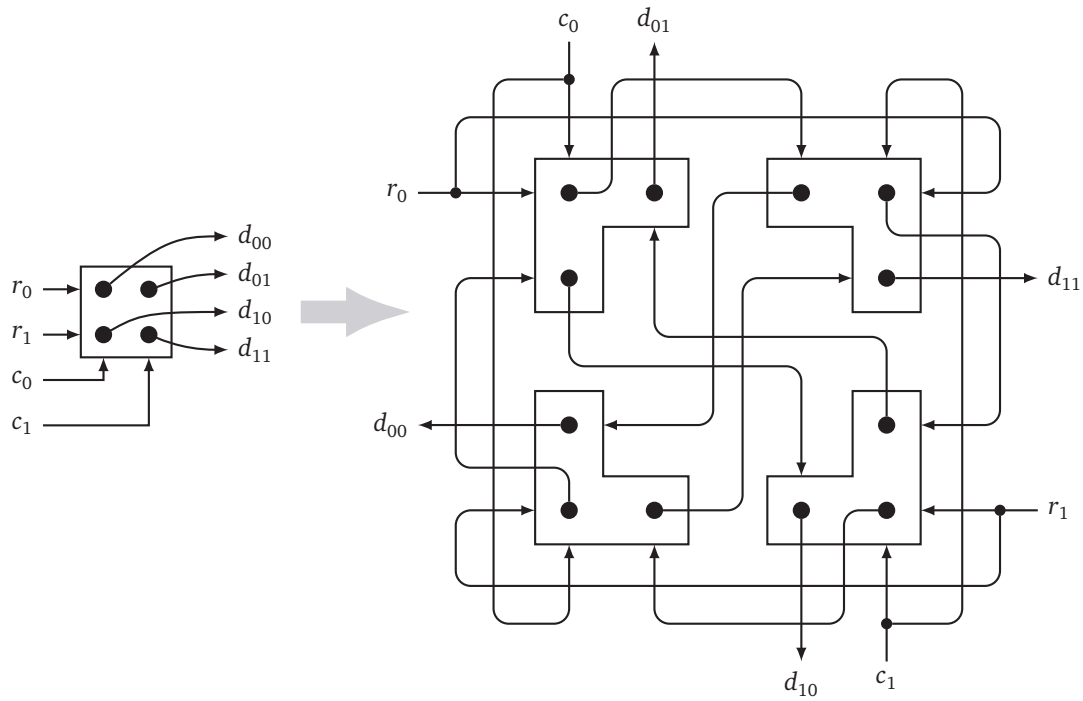


Figure 10.2: a 2-by-2 decision wait using an LJOIN and a FORK as building blocks

each additional column. The intuition is that an input signal to any column but the last sets the corresponding SHUNT, propagates through the TOGGLE below it and then to the left through the MERGE network to synchronize with the row input at the JOIN. The output signal from the JOIN propagates back to the right until it reaches the same SHUNT again, where it gets shunted out through the TOGGLE below it. Alternatively, an input signal to the last column on the right bypasses the SHUNT cascade but still synchronizes with the row input signal at the JOIN, whose output signal passes unimpeded through the whole cascade.

This construction makes economical use of the components, using no more than linearly many of any type, but could also exhibit large latencies when the number of columns is large because a signal has to traverse a number of stages proportional on average to the number of columns. Linear latency may be undesirable and worth trading for other costs if performance is critical, but it is premature to address this issue in depth at this point.

For the moment, we note that a lateral decision wait with n columns is easy to express as Ω_l $n \in \mathbb{H}$ in terms of a function $\Omega_l : \mathbb{N} \rightarrow \mathbb{H}$ defined as follows.

$$\Omega_l(n) = \langle (\lambda c. \mathbf{Z}(\mathbf{F}_c \langle \mathbf{F}(\text{JOIN} : \text{SHUNT}^c), \text{TOGGLE}^c \uparrow_2^1, \text{MERGE } n \rangle \uparrow_1) \uparrow_1) \uparrow_1, \text{JOIN} \rangle_{\delta_1^n} \quad (10.4)$$

That is, letting $c = n - 1$ denote the predecessor of the number of columns, we have the JOIN connected to c instances of a SHUNT

$$\mathbf{F}(\text{JOIN} : \text{SHUNT}^c)$$

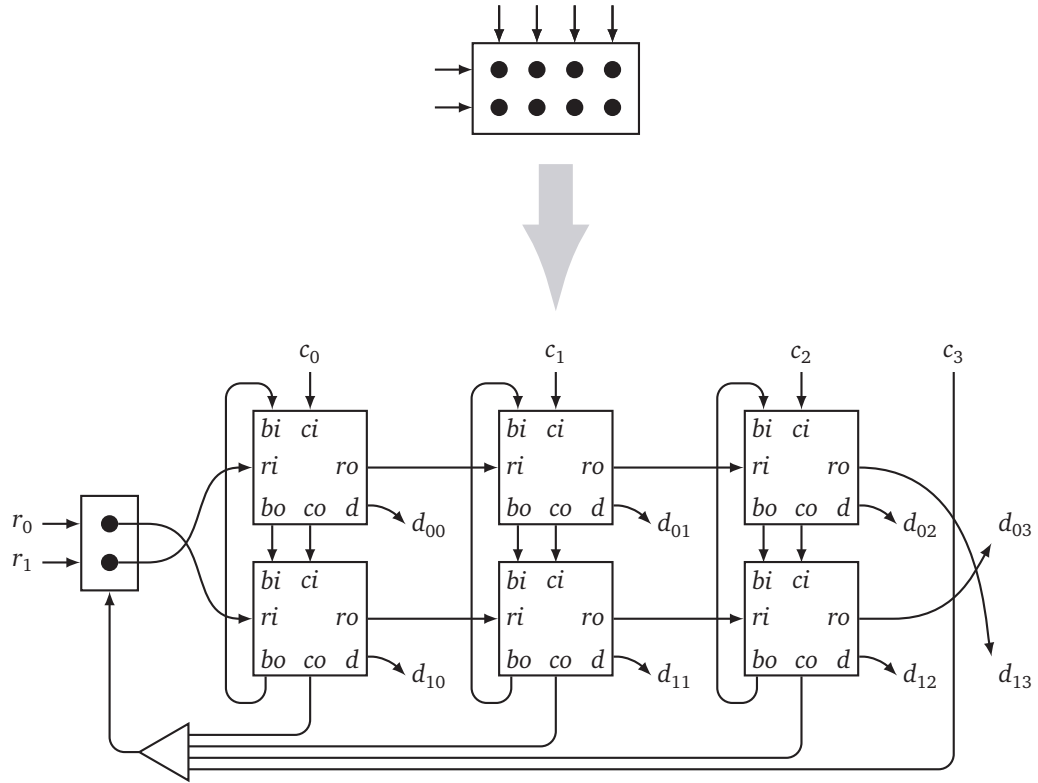


Figure 10.3: A bilateral decision wait made of bilateral decision wait cells has two rows and any number of columns.

a similarly sized cascade $\text{TOGGLE}^c \uparrow_2^1$ with the dotted outputs bundled together, and the network $\text{MERGE } n$ combined in the expression

$$\mathbf{F}_c \langle \mathbf{F}(\text{JOIN} : \text{SHUNT}^c), \text{TOGGLE}^c \uparrow_2^1, \text{MERGE } n \rangle$$

whose last output (from the $\text{MERGE } n$ network) is connected to its first input (to the JOIN) in

$$\mathbf{Z}(\mathbf{F}_c \langle \mathbf{F}(\text{JOIN} : \text{SHUNT}^c), \text{TOGGLE}^c \uparrow_2^1, \text{MERGE } n \rangle \parallel 1)$$

by rolling each of them by one. The case of a 1-by-1 decision wait reduces to a single JOIN primitive, which is indicated separately in [Equation 10.4](#).

10.2.2 Bilateral

The lateral decision wait does not generalize in any obvious way to a bilateral form, so a more creative solution is necessary. One particularly elegant solution shown in [Figure 10.2](#) was reported in [\[222\]](#), but there are reasons to try harder. Although this construction works for a 2-by-2 decision wait, a 2-by- n version of a similar form remains an enigma. Furthermore, at a total cost of 32

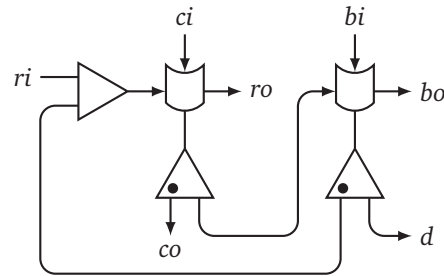


Figure 10.4: a bilateral decision wait cell BC as defined by Equation 10.5

primitives based on the LJOIN implementation in Figure 9.13, perhaps it comes at too high of a price. (In fairness, it would be only 6 primitives if the LJOIN itself were considered a primitive as in [222].)

To design bilateral decision waits with arbitrarily many columns, we fall back instead on the cascading form shown in Figure 10.3, assuming some type of standard cell can be found to make it work. Successful operation is possible via three associated pairs of inputs and outputs on each cell labeled (ci, co) , (ri, ro) , and (bi, bo) , along with the output labeled d visible externally to the cascade.

- Similarly to the lateral decision wait, this network accepts a column input shown from above. On any column but the last, it changes the state of a cell on the top row and then propagates to its neighbor below via the output labeled co .
- A cell on the lower row receives a signal on the input labeled ci , changes its own state accordingly, and relays the signal via co to the MERGE network, from which it reaches the column input on the 2-by-1 decision wait.
- An input signal to the last column bypasses the cells, proceeds directly to the MERGE network, and then synchronizes similarly with the row input.
- After synchronizing with the column input, a row input to the 2-by-1 decision wait propagates to the input labeled ri on the first cell in the opposite row.
- A cell receiving an input signal on ri without previously having received one on ci simply relays it to its neighbor on the right via the output labeled ro .
- A cell receiving an input signal on ri that has previously received one in ci conveys an output signal to bo (to its vertical neighbor) and resumes its original state.
- A cell receiving an input signal on bi (from its vertical neighbor) transmits a signal on the output labeled d and resumes its original state.

A short summary of the required behavior of a typical cell in this cascade is that it can participate in a handshake $ri-ro$ on the terminals thus labeled if it receives no column input signal, $ci-co-ri-bo$ if it receives a column input and then a row input, or $ci-co-bi-d$ if it receives a column input and no

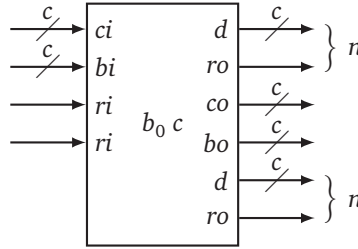


Figure 10.5: A network $b_0 c$ defined by Equation 10.6 contains the repetitive parts of the bilateral decision wait.

row input, before returning to its original state in each case. A design for a cell with this behavior is shown in Figure 10.4 and specified by the expression

$$BC = \mathbf{Z}(\mathbf{U}(\text{MERGE} : (\mathbf{L}(\text{SHUNT}, \text{TOGGLE}))^2) \uparrow 2) \quad (10.5)$$

whose input terminals correspond to those labeled ci , bi , and ri in the figure in that order, and whose outputs correspond to d , ro , co , bo in that order.

To describe the circuit shown in Figure 10.3, let $c = n - 1$ denote the predecessor of the number of columns, so that the expression

$$\mathbf{U}((BC \downarrow 1 \uparrow 2)^c)$$

describes a cascade of c instances of BC by Equation 10.5 each with its output labeled ro connected to the input labeled ri on its neighbor to the right. Affixing an output permutation network

$$\mathbf{F}_{3c} \langle \mathbf{U}((BC \downarrow 1 \uparrow 2)^c), \mathbf{I}^{3c} \times \iota_{3c} \times c \rangle$$

bundles all outputs labeled co , bo and d into three buses of c lines each, in that order, followed by the output labeled ro from the last cell in the cascade. Rolling the inputs up and then connecting an input permutation network

$$\mathbf{F}_{2c} \langle \mathbf{I}^{2c} \leftarrow \iota_2^1, \mathbf{F}_{3c} \langle \mathbf{U}((BC \downarrow 1 \uparrow 2)^c), \mathbf{I}^{3c} \times \iota_{3c} \times c \rangle \uparrow 1 \rangle$$

organizes the inputs labeled ci and bi into two buses of c lines each in that order, followed by the input labeled ri to the first cell. A list containing an instance of this cascade for each of the two rows

$$(\mathbf{F}_{2c} \langle \mathbf{I}^{2c} \leftarrow \iota_2^1, \mathbf{F}_{3c} \langle \mathbf{U}((BC \downarrow 1 \uparrow 2)^c), \mathbf{I}^{3c} \times \iota_{3c} \times c \rangle \uparrow 1 \rangle)^2 \in \mathbb{H}^2$$

leads to a network $b_0 c \in \mathbb{H}$ having the co and bo output buses from the top row connected to the ci and bi buses on the bottom row with $b_0 : \mathbb{N} \rightarrow \mathbb{H}$ defined as follows.

$$b_0 = \lambda c. \mathbf{F}_{2c} \langle (\mathbf{F}_{2c} \langle \mathbf{I}^{2c} \leftarrow \iota_2^1, \mathbf{F}_{3c} \langle \mathbf{U}((BC \downarrow 1 \uparrow 2)^c), \mathbf{I}^{3c} \times \iota_{3c} \times c \rangle \uparrow 1 \rangle)^2 \rangle \quad (10.6)$$

This network has c column inputs followed by c inputs connected to terminals labeled bi on the top row, followed by two row inputs. On the output side, there are n external outputs from the top row, (c labeled d and one more labeled ro) followed by c outputs labeled co from the bottom row,

followed by another c outputs labeled bo from the bottom row, followed by n external outputs from the bottom row as shown in Figure 10.5.

The next step is to connect the bus labeled co on this block to the MERGE network shown in Figure 10.3 while also connecting the bus labeled bo to the one labeled bi and keeping the latter bus lines in the same order. A block $(b_0 c) \uparrow n$ positions both output buses at the beginning so that

$$\mathbf{Z}^{2c} \mathbf{R}((b_0 c) \uparrow n, \mathbf{R}(\mathbf{I}^c, \text{MERGE } n) \downarrow 1)$$

would connect the co bus to the MERGE network, while leaving the bo outputs unconnected but reversed, and one MERGE input free. With the additional rotations

$$(\mathbf{Z}^{2c} \mathbf{R}((b_0 c) \downarrow 2 \uparrow n, \mathbf{R}(\mathbf{I}^c, \text{MERGE } n) \downarrow 1)) \downarrow n$$

the connections from bo to bi can be concluded in

$$\mathbf{Z}^c ((\mathbf{Z}^{2c} \mathbf{R}((b_0 c) \downarrow 2 \uparrow n, \mathbf{R}(\mathbf{I}^c, \text{MERGE } n) \downarrow 1)) \downarrow n)$$

with the original order restored, although this result leaves the output bus labeled d from the bottom row ahead of that of the top. To correct this effect while maintaining the interchanged order of the last ro outputs as shown in Figure 10.3, we can apply an output permutation as shown

$$(\mathbf{Z}^c ((\mathbf{Z}^{2c} \mathbf{R}((b_0 c) \downarrow 2 \uparrow n, \mathbf{R}(\mathbf{I}^c, \text{MERGE } n) \downarrow 1)) \downarrow n)) \times b \langle \iota_c^n, \langle c \rangle, \iota_c, \langle n + c \rangle \rangle$$

and abbreviate this result as $(b_1 b_0) n$ in terms of a function $b_1 : (\mathbb{N} \rightarrow \mathbb{H}) \rightarrow (\mathbb{N} \rightarrow \mathbb{H})$ defined as follows.

$$b_1 = \lambda b. \lambda n. (\lambda c. (\mathbf{Z}^c ((\mathbf{Z}^{2c} \mathbf{R}((b c) \downarrow 2 \uparrow n, \mathbf{R}(\mathbf{I}^c, \text{MERGE } n) \downarrow 1)) \downarrow n)) \times b \langle \iota_c^n, \langle c \rangle, \iota_c, \langle n + c \rangle \rangle) n - 1$$

The block $(b_1 b_0) n$ has two inputs to the terminals labeled ri on the cells in the first column followed by n column inputs (the last being to the MERGE network). On the output side, it has the output from the MERGE network first, followed by the $2n$ external outputs in the correct order. All that remains is to attach the 2-by-1 decision wait shown in Figure 10.3. A 2-by-1 decision wait is expressible in terms of Equation 10.4, and is straightforward to connect to $(b_1 b_0) n$ in an expression

$$\mathbf{F}_2 \langle \Omega_l 2 \downarrow 1, (b_1 b_0) n \rangle$$

where the rows are interchanged as shown in Figure 10.3. The first input to this block is the row input on $\Omega_l 2$ (hence the column input on a 2-by-1 decision wait derived from it) and the first output still comes from the MERGE network. The expression

$$\mathbf{Z}(\mathbf{F}_2 \langle \Omega_l 2 \downarrow 1, (b_1 b_0) n \rangle \uparrow 1)$$

therefore effects the remaining connection from the MERGE network to the decision wait.

Taking one further precaution to allow for the possibility of bilateral decision wait with only one column, which can be given by $\Omega_l 2 \uparrow 1$, we define the bilateral decision wait generating function $\Omega_b : \mathbb{N} \rightarrow \mathbb{H}$ as follows.

$$\Omega_b(n) = \langle \mathbf{Z}(\mathbf{F}_2 \langle \Omega_l 2 \downarrow 1, (b_1 b_0) n \rangle \uparrow 1), \Omega_l 2 \uparrow 1 \rangle_{\delta_l^n} \quad (10.7)$$

10.2.3 General

Upgrading from two rows to any number of rows now requires another round of rumination. The construction in Figure 10.3 works by having a cell in one row tell its neighbor in the other row to clear itself and emit an output. This protocol is appropriate for two rows because there is only one choice for the other row, but does not extend to more than two rows. Any more than one other cell responding similarly would cause multiple outputs to be emitted.

A more promising alternative protocol depends on a cyclic connection among the cells in a column. One cell starts a chain of events by sending a message to its neighbor telling it to clear itself, and the neighbor does not emit an externally visible output but only passes the message along to the next cell in the cycle. When the original cell gets a message back again, the chain of events is complete because the rest of the column is clear, so it clears itself and emits an output.

A column of cells observing this protocol but behaving similarly to the bilateral decision wait cells in other respects would enable a planar decision wait with any number of rows along the lines of Figure 10.6. This diagram may look simpler than Figure 10.3 in that there are no wires crossed between the rows, and it would also work for just two rows, but the bilateral decision wait design previously obtained is still preferable for that case because the bilateral decision wait cell is simpler internally than the planar decision wait cell needed for the more general form. Specifically, the latter must be able to handshake on the terminals labeled *ri-ro* if there is a row input but no column input, *ci-co-pi-po* if there is a column input and no row input, or *ci-co-ri-po-pi-d* otherwise, and then return to its initial state in each case. This operation requires something more like the circuit shown in Figure 10.7 than Figure 10.4.

Planar decision wait cell

To specify this cell, let $s = L\langle \text{SHUNT}, \text{TOGGLE} \rangle$ denote the combination of a SHUNT connected to a TOGGLE that appears three times in it. Taking the two on the right first, we write

$$\mathbf{U}((s \uparrow 1)^2)$$

to express the connection from the middle SHUNT to the one on the right, and then

$$\mathbf{Z}(\mathbf{U}((s \uparrow 1)^2) \parallel 1)$$

to express the connection from the right SHUNT to the middle one. Augmenting this expression with

$$\mathbf{D}_3\langle \mathbf{Z}(\mathbf{U}((s \uparrow 1)^2) \parallel 1), \text{MERGE}^2 \rangle$$

connects the first output from the middle TOGGLE to the first MERGE, the second output from the middle TOGGLE to the second MERGE and the first output from the right TOGGLE also to the second MERGE, which therefore becomes the one whose output is labeled *po*. The output from the first MERGE needs to be connected to the remaining SHUNT, so we roll down the outputs and write

$$\mathbf{U}\langle \mathbf{D}_3\langle \mathbf{Z}(\mathbf{U}((s \uparrow 1)^2) \parallel 1), \text{MERGE}^2 \rangle \downarrow 1, s \rangle$$

to make the connection. A connection is still needed from the last output on the left TOGGLE, which is the last output from this block, to the first input on the middle SHUNT, which is the first input on this block, which we could express by writing

$$\mathbf{Z}(\mathbf{U}\langle \mathbf{D}_3\langle \mathbf{Z}(\mathbf{U}((s \uparrow 1)^2) \parallel 1), \text{MERGE}^2 \rangle \downarrow 1, s \rangle \parallel 1)$$

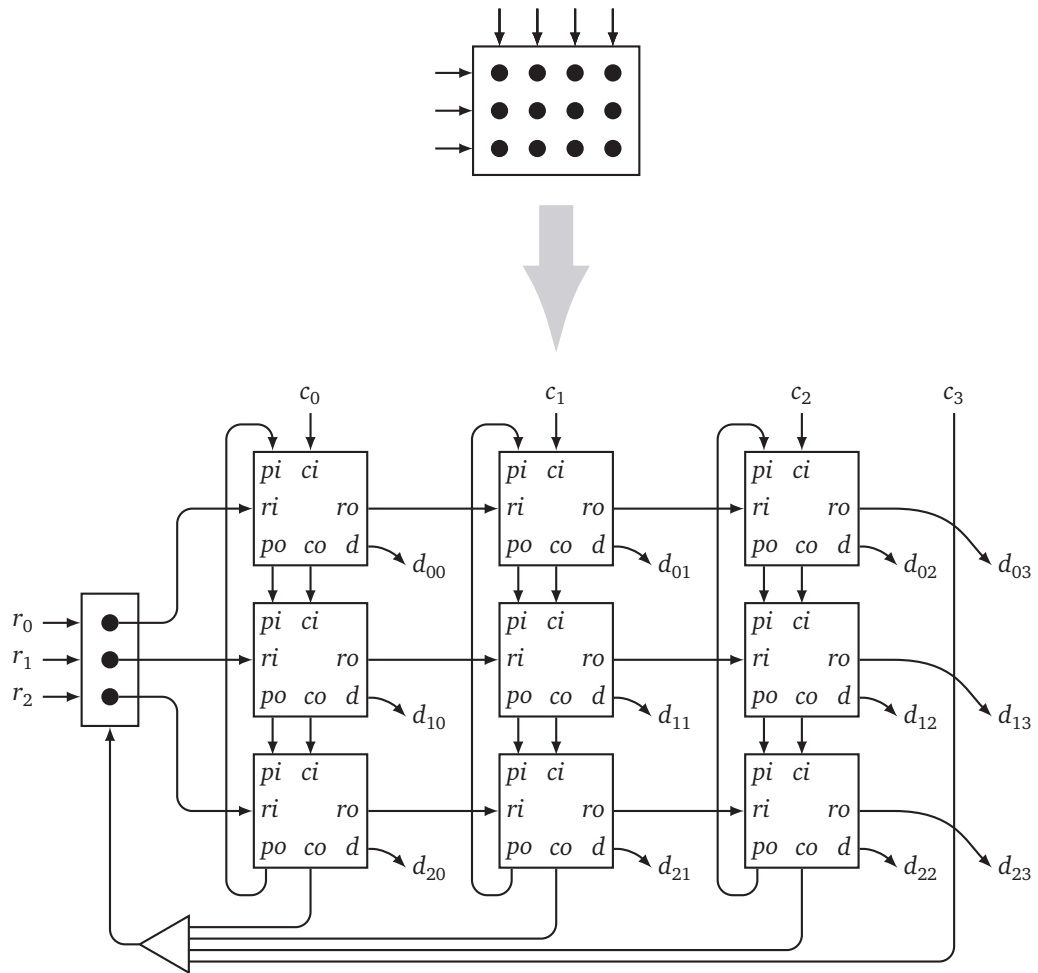


Figure 10.6: A planar decision wait made of planar decision wait cells has any numbers of rows and columns.

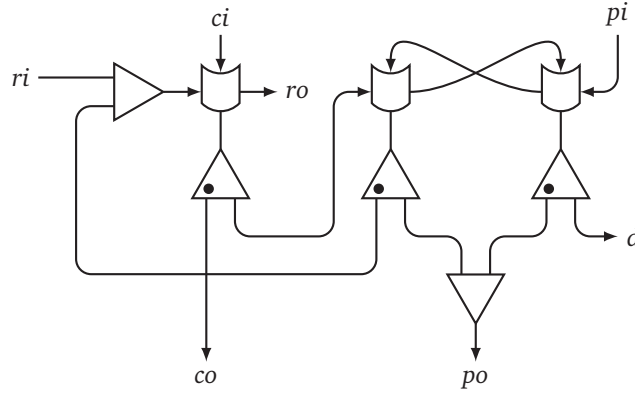


Figure 10.7: planar decision wait cell defined by Equation 10.8

and hence define the planar decision wait cell $PC \in \mathbb{H}$ as

$$PC = (\lambda s. \mathbf{Z}(\mathbf{U}\langle \mathbf{D}_3 \langle \mathbf{Z}(\mathbf{U}((s \uparrow 1)^2) \uparrow 1), \text{MERGE}^2 \rangle \downarrow 1, s \rangle \uparrow 1)) \mathbf{L}\langle \text{SHUNT}, \text{TOGGLE} \rangle \quad (10.8)$$

This block corresponds to Figure 10.7 when the input terminals are labeled pi , ri , and ci in that order, and the outputs are labeled po , d , ro , and co in that order.

Planar decision wait cascade

To describe the cascade in Figure 10.6, we start with a single row of c cells

$$\mathbf{U}((PC \uparrow 1)^\epsilon)$$

where $c = m - 1$ denotes the predecessor of the number of columns m , and the output labeled ro on each cell is connected to the input labeled ri on its neighbor to the right. An additional output permutation network

$$\mathbf{F}_{3c} \langle \mathbf{U}((PC \uparrow 1)^\epsilon), t_{3c} \times c \times \mathbf{I}^{3c} \rangle$$

collects all outputs labeled co , po and d respectively in that order into three buses of c lines each, with a last output labeled ro at the end, and an input permutation network

$$\mathbf{L}_{2c} \langle \mathbf{I}^{2c} \leftarrow \uparrow_2^1, \mathbf{F}_{3c} \langle \mathbf{U}((PC \uparrow 1)^\epsilon), t_{3c} \times 3 \times \mathbf{I}^{3c} \rangle \rangle$$

leaves an input labeled ri at the beginning followed by a bus of c inputs connected to terminals labeled ci and another bus of c inputs connected to terminals labeled pi with respect to Figure 10.6. To make this block easier to connect to those of other rows, we apply two terminal rotations and denote the result $p_0 \in \mathbb{H}$ in terms of a function $p_0 : \mathbb{N} \rightarrow \mathbb{H}$ defined as

$$p_0 = \lambda c. \mathbf{L}_{2c} \langle \mathbf{I}^{2c} \leftarrow \uparrow_2^1, \mathbf{F}_{3c} \langle \mathbf{U}((PC \uparrow 1)^\epsilon), t_{3c} \times 3 \times \mathbf{I}^{3c} \rangle \rangle \uparrow 1 \downarrow c + 1 \quad (10.9)$$

which leads to the block depicted in Figure 10.8 having two buses connected to inputs labeled ci and pi respectively followed by a single input to a terminal labeled ri , an output bus from c terminals

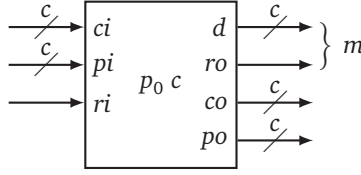


Figure 10.8: A block $p_0 c$ defined by Equation 10.9 contains most of a planar decision wait row.

labeled d , a single output from a terminal labeled ro , and two buses following, each of width c , from terminals labeled co and po respectively.

The next obvious step is to stack the rows by writing

$$\mathbf{U}_{2c}((p_0 c)^n)$$

where n is the number of rows, which connects the co and po labeled outputs from each row respectively to the ci and pi labeled inputs on the row below. The resulting block has c column inputs first, followed by c inputs to terminals labeled pi , followed by n inputs to terminals labeled ri . The output side starts with n buses of m lines each, with each bus consisting of c lines from terminals labeled d followed by one from a terminal labeled ro . Subsequently there are c lines from terminals labeled co and c from terminals labeled po . To close the loop from the po labeled outputs on the bottom row to the pi labeled inputs on the top row, a block

$$\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p_0 c)^n) \parallel c, \mathbf{I}^c)$$

rotates the relevant outputs downward and reverses their order while rotating the corresponding inputs to the end, so that

$$\mathbf{Z}^c(\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p_0 c)^n) \parallel c, \mathbf{I}^c))$$

effects the required connections with their order restored, leaving a block with n row inputs followed by c column inputs, and nm external outputs followed by c outputs from the bottom row terminals labeled co . The addition of a MERGE network

$$\mathbf{L}_c \langle \mathbf{Z}^c(\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p_0 c)^n) \parallel c, \mathbf{I}^c)), \text{MERGE } m \rangle$$

contributes the last required column input while dispensing with the last co labeled outputs. This block lacks only the columnar decision wait shown at the left of Figure 10.6, whose connection to the row inputs is expressible by

$$\mathbf{F}_n \langle \Omega_l n, \mathbf{L}_c \langle \mathbf{Z}^c(\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p_0 c)^n) \parallel c, \mathbf{I}^c)), \text{MERGE } m \rangle \rangle$$

based on Equation 10.4, and whose connection to the MERGE network is expressible by

$$\mathbf{Z}(\mathbf{F}_n \langle \Omega_l n, \mathbf{L}_c \langle \mathbf{Z}^c(\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p_0 c)^n) \parallel c, \mathbf{I}^c)), \text{MERGE } m \rangle \rangle \parallel 1)$$

or more succinctly by $(p_1 p_0)(n, m)$ in terms of a function $p_1 : (\mathbb{N} \rightarrow \mathbb{H}) \rightarrow ((\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{H})$ given by

$$p_1 = \lambda p. \lambda(n, m). (\lambda c. \mathbf{Z}(\mathbf{F}_n \langle \Omega_l n, \mathbf{L}_c \langle \mathbf{Z}^c(\mathbf{Z}^c \mathbf{R}(\mathbf{U}_{2c}((p c)^n) \parallel c, \mathbf{I}^c)), \text{MERGE } m \rangle \rangle \parallel 1)) m - 1. \quad (10.10)$$

Cascading planar decision wait generating function

A generalization of $p_1 p_0$ covering the edge cases of dimensions less than two follows as

$$\Omega_p(n, m) = \langle \langle \langle \langle (p_1 p_0) (n, m), \Omega_b n \uparrow 2 \times \iota_{2n} \times 2 \rangle_{\delta_2^m}, \Omega_l n \uparrow 1 \rangle_{\delta_1^m}, \Omega_b m \rangle_{\delta_2^n}, \Omega_l m \rangle_{\delta_1^n} \rangle. \quad (10.11)$$

That is, for a number of rows $n = 1$ or $n = 2$, it reduces to a lateral or bilateral decision wait respectively with m columns by Equation 10.4 or by Equation 10.7. For any other number of rows and a number of columns $m = 1$ or $m = 2$, it is always better to rotate either a lateral or a bilateral decision wait with n columns into a columnar or bicolunar decision wait respectively with n rows. In all other cases, the cascading planar decision wait $\Omega_p(n, m)$ reduces to $(p_1 p_0) (n, m)$ by Equation 10.9 and Equation 10.10.

Readers who have had enough already could stop here and get by with this definition of a planar decision wait generating function

$$DW(n, m) = \Omega_p(n, m) \quad (10.12)$$

but the more intrepid may find worthwhile improvements on it in subsequent sections.

10.3 Quadrangular decision waits

As noted previously, the cascading planar decision wait suffers from linear latency. An alternative investigated in this section overcomes this limitation by routing the input signals more directly to where they are needed than by traversing the whole sequence of rows and columns. Figure 10.9 shows an example of the kind of thing proposed, which owes a debt to [220], and is called a **quadrangular** decision wait hereafter for lack of a better term.



To reduce clutter in this diagram and others, any network of the form

$$L_n \langle \text{FORK}^n \Gamma_2^1, \text{MERGE } n \rangle$$

can be depicted as exemplified in Figure 10.10 hereafter, and called a 1-hot **completion detecting bus** for the moment. More general discussions of delay insensitive bus protocols and completion detection are deferred to Chapter 13.

The theory of operation can be summarized roughly as follows by using Figure 10.9 as an example. This quadrangular decision wait lets any of the first i row inputs enable the top row of the bilateral decision wait routing stage by way of a completion detecting bus, and any of the latter j row inputs enable the other row. The bilateral routing stage thereby routes the column inputs either to the upper or lower level of the array of internal decision waits accordingly. A similar mechanism routes the row inputs to either the left or the right side of the array of internal decision waits, so that the net effect is for exactly one of the internal decision waits to receive a row and a column input signal, and the rest to receive none. A permutation network (not shown) rearranges the output lines from the four internal decision waits to an order consistent their respective roles as one quarter of the whole.

A hand waving argument for this construction being faster than the cascading form in the limit of large sizes is that its latency is determined by that of a decision wait only one quarter of its

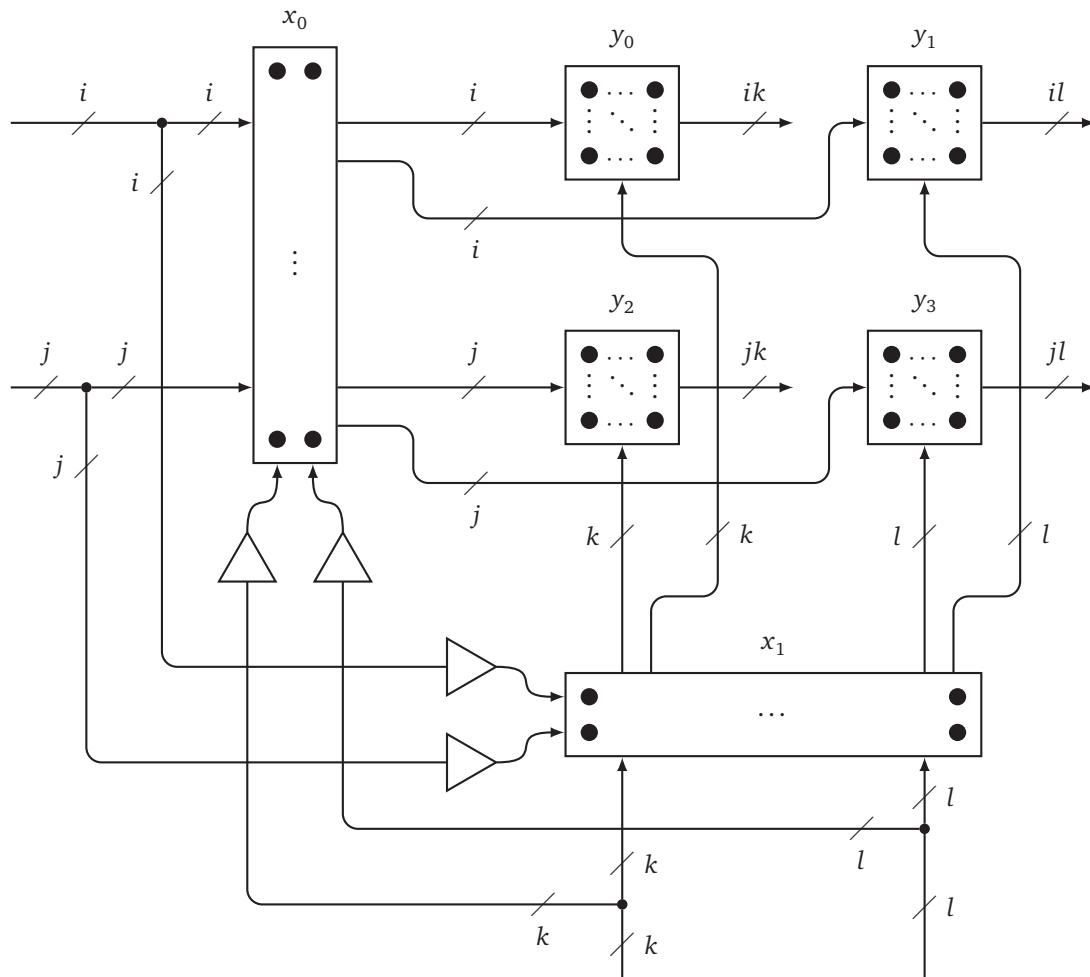
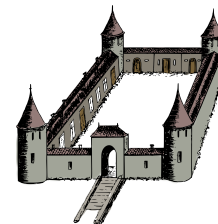


Figure 10.9: A quadrangular decision wait with dimensions $\langle\langle i, j \rangle, \langle k, l \rangle\rangle$ all greater than 1 has an array of four internal decision waits y_0 through y_3 , a bicolunar row input routing stage x_0 , a bilateral column input routing stage x_1 , and four completion detecting buses.

size plus whatever time the input signals take to propagate through the front end routing stages, which might be similarly decomposed. This total is plausibly logarithmic rather than linear in the dimensions. An exact critical path analysis can be found in [Section C.2](#).



The rest of this section is devoted to the formal specification of quadrangular decision waits. A basic construction in [Section 10.3.1](#) covers any configuration, while that of [Section 10.3.2](#) reduces the cost for the special case of a columnar array of internal decision waits y_i . [Section 10.3.3](#) adapts the latter to lateral arrays and subsumes these three alternatives along with the degenerate case of a 1-by-1 array into a general form.

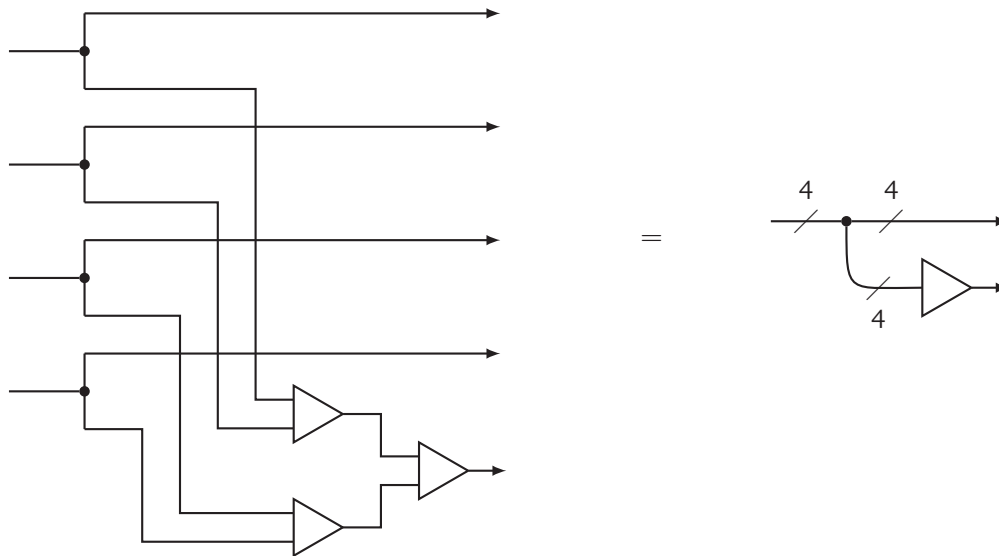


Figure 10.10: schematic abbreviation of a 4-line 1-hot bus with completion detection

10.3.1 Basic

A quadrangular decision wait need not be limited to a 2-by-2 array of internal decision waits as shown in Figure 10.9, but may contain an array of any number of them provided their dimensions match up. That is, all decision waits in the array must have the same number of rows as their neighbors to the sides, and the same number of columns as their neighbors above and below. When there are more than two across, the routing stage at the left needs more than two columns, and when there are more than two in the vertical direction, the routing stage below needs more than two rows.

A concise account of all relevant dimensions therefore can be given by a list $d \in \mathbb{N}^{*2}$ such that $|d_0|$ is the number of decision waits along the vertical dimension of the array, $|d_1|$ is the number along the horizontal dimension, d_{0i} is the number of rows of the decision waits in the i -th vertical position of the array, and d_{1j} is the number of columns common to all decision waits in the j -th horizontal position of the array. It also follows that

- the routing stage for the row inputs has dimensions $\sum d_{0\text{-by-}}|d_1|$
- the other routing stage has dimensions $|d_0|\text{-by-}\sum d_1$
- the number of decision waits in the internal array is $|d_0||d_1|$
- and the dimensions of the whole result are $\sum d_{0\text{-by-}}\sum d_1$.

It might seem appropriate at this point to seek a function $\ddot{\Omega}_q$ analogous to Ω_p that takes the list $d \in \mathbb{N}^{*2}$ to a quadrangular decision wait with these dimensions, but d does not fully determine the quadrangular decision wait if we want to allow complete flexibility in the decompositions of its constituent parts. Ideally each routing stage and each member of the array might be a smaller

quadrangular decision wait specified by a separate list d of its own or might be of the form $\Omega_p(n, m)$. Hence we take a broader view by thinking of $\ddot{\Omega}_q$ only as a *combining form*

$$\ddot{\Omega}_q : \mathbb{N}^{*2} \times \mathbb{H}^2 \times \mathbb{H}^* \rightarrow \mathbb{H}$$

such that $\ddot{\Omega}_q(d, x, y) \in \mathbb{H}$ refers to a quadrangular decision wait with given dimensions d , given routing stage decision waits $x \in \mathbb{H}^2$, and given internal decision waits $y \in \mathbb{H}^{|d_0|+|d_1|}$. By way of further convention, we construct $\ddot{\Omega}_q$ based on x_0 being the stage with dimensions $\sum d_0$ -by- $|d_1|$ and x_1 being the stage with dimensions $|d_0|$ -by- $\sum d_1$. The members of y are taken to be listed in “row major” order, which is to say that $y_{i|d_0|+j}$ has dimensions d_{0i} -by- d_{1j} for $0 \leq i < |d_0|$ and $0 \leq j < |d_1|$.

Completion detecting buses

To work our way toward a formal definition of $\ddot{\Omega}_q$ from the front inwards, we can start by describing the completion detecting buses in the quadrangular decision wait as $(\mathcal{F} \mathbf{R}) q_0^* d \in \mathbb{H}$ in terms of a function $q_0 : \mathbb{N}^* \rightarrow \mathbb{H}$ taking a list of dimensions $a \in \mathcal{R}(d)$ to a network of $|a|$ completion detecting buses. Each term $h \in \mathcal{R}(a)$ calls for a network

$$\mathbf{L}_h \langle \text{FORK}^h \mapsto \frac{1}{2}, \text{MERGE } h \rangle$$

suggesting a folded function over a

$$(\mathcal{F}_{\mathbf{Z}1} \lambda(h, t) \cdot \mathbf{R}(\mathbf{L}_h \langle \text{FORK}^h \mapsto \frac{1}{2}, \text{MERGE } h \rangle \downarrow 1, t) \uparrow 1) a$$

resulting in a network of $|a|$ completion detecting buses with their MERGE outputs segregated for easier connection to the opposite routing stage. This expression unfortunately reverses the order of the MERGE outputs relative to their respective buses, so a definition of q_0 correcting for this effect is preferable.

$$q_0 = \lambda a. \mathbf{Z}^{|a|} \mathbf{R}(((\mathcal{F}_{\mathbf{Z}1} \lambda(h, t) \cdot \mathbf{R}(\mathbf{L}_h \langle \text{FORK}^h \mapsto \frac{1}{2}, \text{MERGE } h \rangle \downarrow 1, t) \uparrow 1) a) \downarrow |a|, |a|) \quad (10.13)$$

Front permutation network

Next we envision the completion detecting bus network $(\mathcal{F} \mathbf{R}) q_0^* d$ connected to the parallel combination of input stages $\mathbf{R}(x_0, x_1)$ by a permutation network described by a permutation as in

$$(\mathcal{F} \mathbf{R}) q_0^* d \xrightarrow{q_1 d} \mathbf{R}(x_0, x_1)$$

for some function $q_1 : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$. The permutation network is fully determined by its input coming from four buses of widths $\sum d_0$, $|d_0|$, $\sum d_1$ and $|d_1|$ in that order and its output going to four buses of these same widths but in the order $\sum d_0$, $|d_1|$, $|d_0|$ and $\sum d_1$. The latter order results from the row inputs of x_0 followed by the column inputs of x_0 , both followed by the row inputs of x_1 with the column inputs of x_1 last. These conditions imply the following definition for q_1 .

$$q_1 = \lambda d. \mathbf{b} \langle \iota_{\sum d_0}, \iota_{|d_0|}^{|d_1|+\sum d_0}, \iota_{\sum d_1}^{|d_0|+|d_1|+\sum d_0}, \iota_{|d_1|}^{\sum d_0} \rangle \quad (10.14)$$

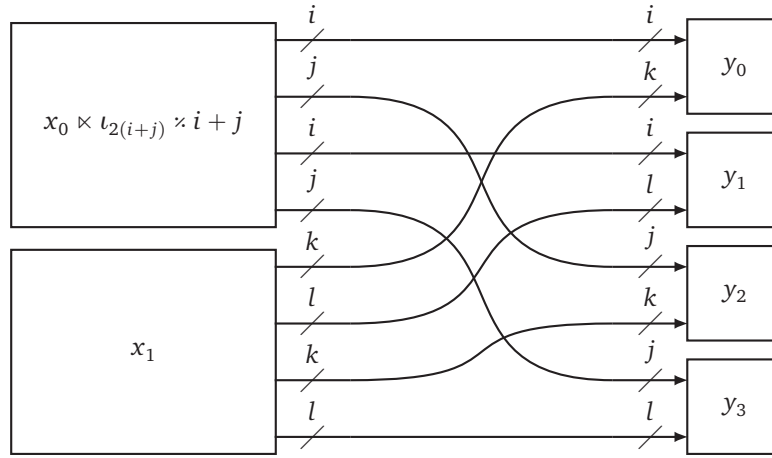


Figure 10.11: the central permutation network given by $q_2 d$ for a basic quadrangular decision wait with dimensions $d = \langle\langle i, j \rangle, \langle k, l \rangle\rangle$

Central permutation network

Continuing inward, next we need a permutation network to connect the routing stages to the internal array $(\mathcal{F} \mathbf{R}) y \in \mathbb{H}$ according to a permutation $q_2 d$ in terms of a function $q_2 : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$. This specification would be complicated by the requirement for non-consecutive outputs from x_0 to connect to consecutive inputs on terms of y , but could be simplified if x_0 were replaced by

$$x_0 \times \iota_{|d_1| \sum d_0} \times \sum d_0$$

thereby bundling each of the $|d_1|$ columns of outputs from x_0 into a separate bus of width $\sum d_0$ in

$$(\mathcal{F} \mathbf{R}) q_0^* d \xrightarrow{q_1 d} \mathbf{R}(x_0 \times \iota_{|d_1| \sum d_0} \times \sum d_0, x_1) \xrightarrow{q_2 d} (\mathcal{F} \mathbf{R}) y.$$

No changes to Equation 10.14 are needed because the input order of x_0 is unaffected.

Even so, a formula for q_2 is not obvious. Reading the permutation in Figure 10.11 from Figure 10.9 suggests no clear generalization to arbitrary dimensions. To break it down a bit further, let

$$p = \flat (\lambda r. (\lambda c. \langle r, c \rangle)^* d_1)^* d_0 \in (\mathbb{N}^2)^{|d_0| |d_1|} \tag{10.15}$$

denote the list of dimensions of the items of y , with each $p_n = \langle r, c \rangle \in \mathbb{N}^2$ expressing the number of rows r and the number of columns c in the n -th item of y . Then the value of p pertaining to Figure 10.11 would be $\langle\langle i, k \rangle, \langle i, l \rangle, \langle j, k \rangle, \langle j, l \rangle\rangle$, for example, and the permutation might be more easily described in reference to p .

The permutation consists of two concatenated parts, the first accounting for the connections from x_0 to the row inputs in y , and the second accounting for the connections from x_1 to the column inputs in y . Each part has one sublist for each term of p . To consider the row inputs first, a typical bundle of them, such as the n -th, is carried by a bus of width p_{n0} , corresponding to r in the list $\langle r, c \rangle$ noted above. If this bus is connected to the n -th decision wait in y , then the initial line in this bus

connects to the input terminal numbered $\sum^b(p \mid n)$ on $(\mathcal{F} \mathbf{R}) y$, the total number of inputs on all decision waits in $(\mathcal{F} \mathbf{R}) y$ preceding the n -th. We could almost write

$$b(\lambda n. \iota_{p_{n0}}^{\sum^b(p \mid n)})^* \iota_{|p|}$$

to put all of the line numbers together were it not for the fact that the n -th bundle of row inputs to y does not necessarily come from the n -th position relative to $x_0 \times \iota_{|d_1| \sum d_0} \times \sum d_0$. Taking a cue from [Figure 10.11](#), we note that the output bus ordering is permuted further in addition to the output permutation already associated with x_0 . A simple transpose by $|d_0|$ accounts for this effect, making

$$b(((\lambda n. \iota_{p_{n0}}^{\sum^b(p \mid n)})^* \iota_{|p|}) \times |d_0|)$$

the correct expression of this part of the permutation.

The second part of the permutation is more straightforward because the outputs from x_1 are already in an order that matches the column input buses on $(\mathcal{F} \mathbf{R}) y$. The n -th term in y has a number of column inputs p_{n1} , whose first is on the terminal numbered $p_{n0} + \sum^b(p \mid n)$ relative to $(\mathcal{F} \mathbf{R}) y$, which is the total number of inputs due to preceding terms plus the number of row inputs on the n -th. The list of all of the input terminal numbers for column inputs to y therefore would be

$$b(\lambda n. \iota_{p_{n1}^{p_{n0} + \sum^b(p \mid n)}}^*)^* \iota_{|p|}$$

and the entire permutation $q_2 d$ therefore given by

$$q_2 = \lambda d. (\lambda p. b(((\lambda n. \iota_{p_{n0}}^{\sum^b(p \mid n)})^* \iota_{|p|}) \times |d_0| \parallel (\lambda n. \iota_{p_{n1}^{p_{n0} + \sum^b(p \mid n)}}^*)^* \iota_{|p|})) b(\lambda r. (\lambda c. \langle r, c \rangle)^* d_1)^* d_0.$$

Back permutation network

One last consideration is a permutation network to order the outputs from $(\mathcal{F} \mathbf{R}) y$ as if they came from a single decision wait with dimensions $\sum d_0$ -by- $\sum d_1$. Without an output permutation network, all of the outputs from the first decision wait y_0 would be first, followed by all of the outputs from y_1 , and so on, and within those of each decision wait, the outputs would emerge row by row. The correct order takes one row from each of the first $|d_1|$ decision waits y_0 through $y_{|d_1|-1}$, and then the next row from each of the first $|d_1|$, until all rows of all decision waits y_0 through $y_{|d_1|-1}$ are taken, and then moves on to first row of the next group of $|d_1|$ decision waits, and so on.

A formal description of this ordering starts with a row from a typical decision wait in the j -th horizontal position of the array, which is carried by a bus of width d_{1j} .

- If this bus were from the top row of one of the first $|d_1|$ decision waits, then its first line would reach a position numbered $\sum(d_1 \mid j)$ relative to all other bus lines, which is the sum of the bus widths due to the first rows of the decision waits preceding it in the array.
- If this bus were from the k -th row of one of the first $|d_1|$ decision waits, the destination position of its first line would be offset additionally by $k \sum d_1$, the total widths of all buses from the rows above.
- If the bus came not necessarily from one of the first $|d_1|$ decision waits in the array but from the i -th group of them, then the destination of its first line would be further offset by the total widths of all buses due to all groups of decision waits above it in the array, $(\sum(d_0 \mid i)) \sum d_1$.

In general, a bus of d_{1j} lines coming from the k -th row of the j -th decision wait in the i -th group of $|d_1|$ decision waits has an offset of

$$(k + \sum(d_0 \uparrow i))(\sum d_1) + \sum(d_1 \uparrow j)$$

relative to the other bus lines. Any decision wait in the i -th group has d_{0i} rows, so the full list of destination positions for outputs from this typical decision wait would be

$$b(\lambda k. \iota_{d_{1j}}^{(k + \sum(d_0 \uparrow i))(\sum d_1) + \sum(d_1 \uparrow j)})^* \iota_{d_{0i}}$$

those of the whole group would be

$$b^2(\lambda j. (\lambda k. \iota_{d_{1j}}^{(k + \sum(d_0 \uparrow i))(\sum d_1) + \sum(d_1 \uparrow j)})^* \iota_{d_{0i}})^* \iota_{|d_1|}$$

and those of the whole array would be $q_3 d$ for $q_3 : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$ given by

$$q_3 = \lambda d. b^3(\lambda i. (\lambda j. (\lambda k. \iota_{d_{1j}}^{(k + \sum(d_0 \uparrow i))(\sum d_1) + \sum(d_1 \uparrow j)})^* \iota_{d_{0i}})^* \iota_{|d_1|})^* \iota_{|d_0|}.$$

The list $q_3 d$ as defined above can be viewed as a function that maps an output terminal number on the block $(\mathcal{F} \mathbf{R}) y$ to the corresponding output terminal number on the quadrangular decision wait $\tilde{\Omega}(d, x, y)$. Using it correctly in a formal definition of $\tilde{\Omega}$ requires inverting it to maintain the convention stipulated along with Equation 8.53, whereby output permutations map external terminal numbers to internal ones. Hence we define $\tilde{\Omega}_q$ as follows.

$$\tilde{\Omega}_q(d, x, y) = (\mathcal{F} \mathbf{R}) q_0^* d \xrightarrow{q_1 d} \mathbf{R}(x_0 \times \iota_{|d_1| \sum d_0} \times \sum d_0, x_1) \xrightarrow{q_2 d} (\mathcal{F} \mathbf{R}) y \times (q_3 d)^{-1} \quad (10.16)$$

10.3.2 Vertical

Quadrangular decision waits built according to Equation 10.16 work for any valid choice of parameters d , x , and y , but are suboptimal for some. For example, if $d_1 = \langle d_{10} \rangle$ had only one item, then y would be arranged in an array of just one column, leaving no need for an explicit $|d_0|$ -by-1 routing stage x_0 or for the completion detector network $q_0 d_1$. Instead, the whole result could be constructed as shown in Figure 10.12. Further economies would be possible if any term d_{0i} were equal to 1, because then the 1-by- d_{10} lateral decision wait y_i could be omitted from the result in favor of taking the corresponding d_{10} outputs directly from the i -th row of the routing stage x_1 as shown in Figure 10.13.

Constructing a quadrangular decision wait to exploit both of these optimizations under the assumption of a unit list d_1 is best done mostly from scratch. One of the things needed repeatedly for this job is a list of two functions

$$l = \langle \lambda a. a \uparrow \mathbb{N} - \{1\}, \lambda a. |a \uparrow \{1\}| \rangle \in ((\mathbb{N}^* \rightarrow \mathbb{N}) \cup (\mathbb{N}^* \rightarrow \mathbb{N}^*))^2$$

such that $l_0 : \mathbb{N}^* \rightarrow \mathbb{N}^*$ takes a list of dimensions $a \in \mathbb{N}^*$ to the list $l_0 a \in \mathbb{N}^*$ derived from a by deleting all items equal to 1 from it (by Equation 8.8), and $l_1 : \mathbb{N}^* \rightarrow \mathbb{N}$ takes a list of dimensions $a \in \mathbb{N}^*$ to the number $l_1 a$ of items equal to 1 in it.

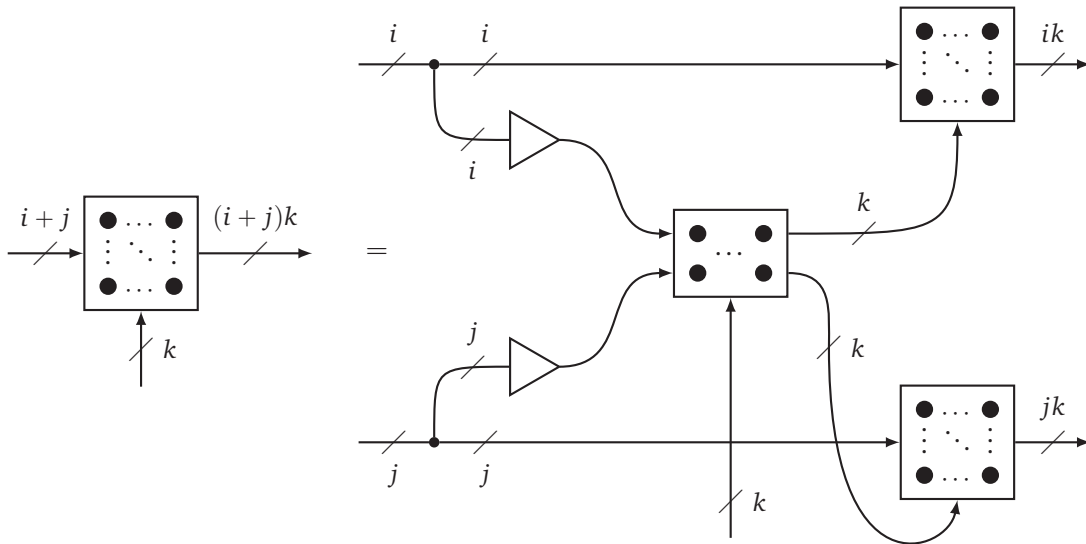


Figure 10.12: a quadrangular decision wait with dimensions $\langle\langle i, j \rangle, \langle k \rangle\rangle$ with i and j both greater than 1

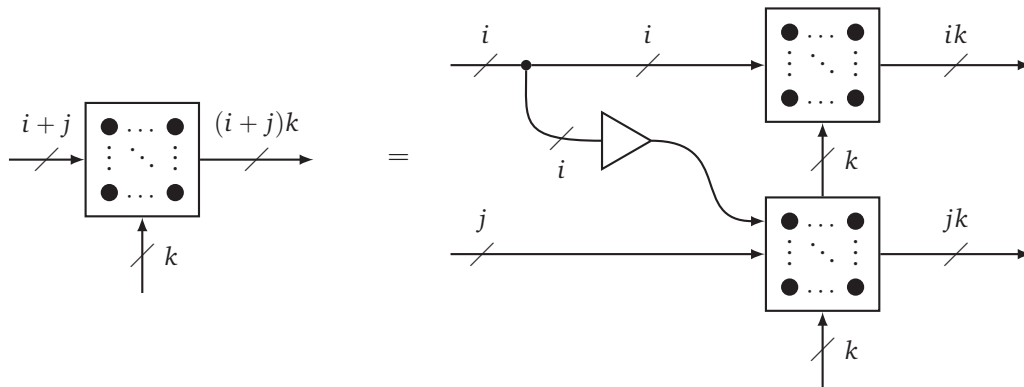


Figure 10.13: a quadrangular decision wait with dimensions $\langle i : 1^j, \langle k \rangle \rangle$ and $i, j > 1$

Front end permutations

The first use for l is in the specification of an input permutation network that interfaces $\sum l_0 d_0$ of the $\sum d_0$ external row inputs with a completion detecting bus network $q_0 l_0 d_0$ by Equation 10.13, whose MERGE outputs connect to the row inputs on x_1 as in the basic construction, but connects the remaining $l_1 d_0$ external row inputs directly to the row inputs on x_1 as shown in Figure 10.13. The permutation would look something like

$$b(\lambda i. \langle \iota_{d_{0i}}^{\sum l_0(d_0 \mid i)}, \langle (\sum l_0 d_0) + l_1(d_0 \mid i) \rangle \rangle_{\delta_1^{d_{0i}}} \rangle^* \iota_{|d_0|})$$

so that each term d_{0i} greater than 1 would induce an input bus of width d_{0i} ordered among those of similar terms, but any unit term $d_{0i} = 1$ would indicate a destination position offset by $\sum l_0 d_0$ in addition to the number $l_1(d_0 \mid i)$ of preceding unit terms. A network described by this permutation connected to the front of a block $\mathbf{R}(q_0 l_0 d_0, l_1^{d_0})$ would route each of the $\sum d_0$ external row inputs to the completion detecting bus network $q_0 l_0 d_0$ or not as appropriate, but the block's last $l_1 d_0$ outputs might not match the row input order of x_1 without another permutation network of a related form between them. A permutation describing the latter would look something like

$$(\lambda i. \langle |l_0(d_0 \mid i)|, |l_0 d_0| + l_1(d_0 \mid i) \rangle \rangle_{\delta_1^{d_{0i}}} \rangle^* \iota_{|d_0|})$$

for the similar reason that the MERGE outputs would form a separate sequence from those associated with the unit inputs. Because these two permutations have much in common, we can express both of them at once more succinctly as

$$\langle p_0, p_1 \rangle = b^*(((\lambda i. \langle \iota_{a_i}^{\sum l_0 r}, \langle |l_0 r| \rangle \rangle, \langle (\sum l_0 a) + l_1 r \rangle, \langle |l_0 a| + l_1 r \rangle \rangle)_{\delta_1^{a_i}} \rangle^* \iota_{|a|}) \times |a|)$$

where $a = d_0$ is the list of row dimensions and $r = d_0 \mid i$ is the list of row dimensions preceding the i -th, or more formally as $p = q_4(l, d_0)$ for a function

$$q_4 : ((\mathbb{N}^* \rightarrow \mathbb{N}) \cup (\mathbb{N}^* \rightarrow \mathbb{N}^*))^2 \times \mathbb{N}^* \rightarrow \mathbb{N}^{*2}$$

defined by

$$q_4 = \lambda(l, a). b^*(((\lambda i. (\lambda r. \langle \iota_{a_i}^{\sum l_0 r}, \langle |l_0 r| \rangle \rangle, \langle (\sum l_0 a) + l_1 r \rangle, \langle |l_0 a| + l_1 r \rangle \rangle)_{\delta_1^{a_i}} \rangle^* \iota_{|a|}) \times |a|).$$

Front end stages

As intended, we can now incorporate the routing stage x_1 , the completion detecting bus network $q_0 l_0 d_0$ and the external row inputs correctly into a block $(q_5 \langle q_0, q_4 \rangle)(l, d_0, x_1)$ for a straightforward choice of

$$q_5 = \lambda q. \lambda(l, a, x). (\lambda p. \mathbf{U}_{|a|} \langle p_0 \times \mathbf{R}(q_0 l_0 a, l_1^a) \times p_1, x \rangle) q_1(l, a).$$

Only one of the routing stages x_1 is used, because x_0 can be ignored based on the assumption of a unit list of column dimensions d_1 . Note that q_1 is instantiated as q_4 in the context of a formal parameter $q = \langle q_0, q_4 \rangle$.

Central permutation

Whereas the block above completes the front end, the back end consists of an array of decision waits selected from y . As noted previously, this array should omit the lateral terms in y as an optimization, which are the terms y_i for which d_{0i} is unity, leaving only the remaining $|l_0 d_0|$ terms $y^* l_0 d_0$, which have a total of

$$\sum (q_4(l, d_0)_0^{-1} \mid |l_0 d_0|)$$

row inputs interspersed with $d_{10}|l_0 d_0|$ column inputs in buses of $d_{10} = \sum d_1$ lines each.

The number of surviving terms in y matches the number of completion detecting buses in the front end by design, so a specification for a permutation network connecting the FORK outputs from the front end to the row inputs in this array could take the form

$$b \left(\lambda i. t_{(q_4(l, d_0)_0^{-1})_i}^{i d_{10} + \sum ((q_4(l, d_0)_0^{-1})_i)} \right)^* t_{|l_0 d_0|}$$

when concatenated with one that describes the connections from the $d_{10} \sum d_0$ outputs from x_1 to the column inputs. This total generally exceeds the number of column inputs $d_{10}|l_0 d_0|$ in the array due to the $l_1 d_0$ lateral terms omitted from y , so we envision the array in parallel with $l_1 d_0$ extra buses each of width d_{10} to reconcile the front and back end arities.

$$\mathbf{R}((\mathcal{F} \mathbf{R}) y^* l_0 d_0, l^{d_{10}(l_1 d_0)})$$

The i -th row of x_1 then connects by a bus of width d_{10} either to the j -th bundle of column inputs in the array for $j < |l_0 d_0|$, or to the $(j - |l_0 d_0|)$ -th extra bus at the end, where j is given by $q_4(l, d_0)_{1i}$ according to the same permutation used to map the MERGE outputs and some of the external row inputs to the row inputs of x_1 . In either case, the first line from this bus reaches a terminal on the back end whose position is offset by the $j d_{10}$ column inputs preceding it, plus the

$$\sum ((q_4(l, d_0)_0^{-1}) \mid j)$$

row inputs preceding it, suggesting the rest of the desired permutation network specification.

$$b \left(\lambda i. (\lambda j. t_{d_{10}}^{j d_{10} + \sum ((q_4(l, d_0)_0^{-1})_i)} \right) q_4(l, d_0)_{1i} \right)^* t_{|d_0|}$$

Concatenating, we have

$$\left(b \left(\lambda i. t_{(q_4(l, d_0)_0^{-1})_i}^{i d_{10} + \sum ((q_4(l, d_0)_0^{-1})_i)} \right)^* t_{|l_0 d_0|} \right) \parallel \left(b \left(\lambda i. (\lambda j. t_{d_{10}}^{j d_{10} + \sum ((q_4(l, d_0)_0^{-1})_i)} \right) q_4(l, d_0)_{1i} \right)^* t_{|d_0|} \right)$$

or more briefly $(q_6 q_4)(l, d)$ with q_6 given by

$$q_6 = \lambda q. \lambda(l, d). (\lambda p. b \left(((\lambda i. t_{(p_0^{-1})_i}^{i d_{10} + \sum (p_0^{-1})_i)} \right)^* t_{|l_0 d_0|} \parallel ((\lambda i. t_{d_{10}}^{p_{1i} d_{10} + \sum (p_0^{-1})_i)} \right)^* t_{|d_0|} \right))) q(l, d_0)$$

after some routine simplification.

Back end

The back end $\mathbf{R}((\mathcal{F} \mathbf{R}) y^* l_0 d_0, l^{d_{10}(l_1 d_0)})$ assumed above contains $|l_0 d_0|$ decision waits followed by $l_1 d_1$ buses, whose outputs may need rearrangement. Specifically, the outputs from the i -th decision wait or bus should appear externally in the j -th position relative to those of the others, where

$$j = (q_4(l, d_0)_1^{-1})_i$$

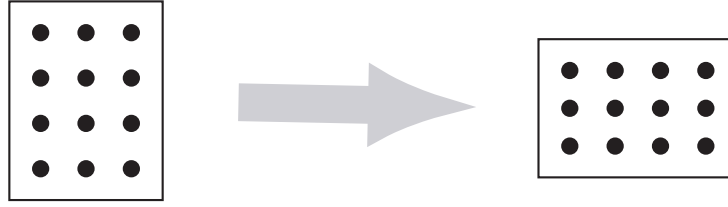


Figure 10.14: Rotating a 4-by-3 decision wait turns it into a 3-by-4 decision wait.

is determined by the permutation applied to the rows on the front end. These outputs would be carried by a bus of width d_0, d_{10} whose first line attains a position offset by $d_{10} \sum (d_0 + j)$, the number of outputs preceding it. A permutation mapping all back end outputs to their correct external positions therefore could be given by

$$b (\lambda i. (\lambda j. t_{d_{10} d_0}^{d_{10} \sum (d_0 + j)}) (q_4(l, d_0)_1^{-1})_i)^* t_{|d_0|}$$

and the adjusted back end by $(q_7 q_4) (l, d, y)$ for a function q_7 defined as

$$q_7 = \lambda q. \lambda (l, d, y). \mathbf{R}((\mathcal{F} \mathbf{R}) y^* l_0 d_0, \mathbf{I}^{d_{10}(l_1 d_0)}) \times (b (\lambda i. (\lambda j. t_{d_{10} d_0}^{d_{10} \sum (d_0 + j)}) (q(l, d_0)_1^{-1})_i)^* t_{|d_0|})^{-1}$$

where the output permutation is inverted per convention. As a result, we have the following definition for a vertical quadrangular decision wait combining form

$$\dot{\Omega}_q(d, x, y) = (\lambda l. (q_5 \langle q_0, q_4 \rangle) (l, d_0, x_1) \xrightarrow{(q_6 q_4) (l, d)} (q_7 q_4) (l, d, y)) \langle \lambda a. a \uparrow \mathbb{N} - \{1\}, \lambda a. |a \uparrow \{1\}| \rangle$$

equivalent to $\ddot{\Omega}_q$ only for $|d_1| = 1$, but more efficient in that case.

10.3.3 General

Even better than $\dot{\Omega}_q$ and $\ddot{\Omega}_q$ would be a quadrangular decision wait combining form that always generates the preferable result for the given dimensions whether vertical or not, and perhaps also does something smarter for the horizontal case as well. These matters are addressed briefly in this section.

Horizontal quadrangular decision waits

The latter problem is solvable without deriving a horizontal analog to $\dot{\Omega}_q$ *ab initio*. Rotating each constituent decision wait of its parameters and a few other tweaks make it useful for the horizontal case. More general decision wait transformations are discussed in Section 10.5, but for a sneak preview, the simple form of rotation restricted to planar decision waits as illustrated in Figure 10.14 is captured by a function $q_8 : \mathbb{N} \times \mathbb{N} \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ defined by

$$q_8 = \lambda (r, c). \lambda p. p \uparrow r \times t_{rc} \times r \tag{10.17}$$

which takes any pair of dimensions $(r, c) \in \mathbb{N} \times \mathbb{N}$ to a function $q_8(r, c) : \mathbb{H} \rightarrow \mathbb{H}$ that takes any r -by- c decision wait $p \in \mathbb{H}$ to a c -by- r decision wait $q_8(r, c) p \in \mathbb{H}$. This transformation amounts

to interchanging the row inputs with the column inputs while transposing the output order (cf. Equation 10.11).

A horizontal quadrangular decision wait is determined by a triple of parameters

$$(d, x, y) \in \mathbb{N}^{*2} \times \mathbb{H}^2 \times \mathbb{H}^{|d_0||d_1|}$$

with $|d_0| = 1$, but we can use $\dot{\Omega}_q$ nevertheless to achieve a better result than $\ddot{\Omega}_q(d, x, y)$ by applying it to parameters $\langle d_1, d_0 \rangle$ in place of d and rotating x_0 and the terms of y . That is, by using $q_8(d_{00}, \sum d_1) x_0$ in place of x_1 and $q_8(d_{00}, d_{1i}) y_i$ in place of each y_i we obtain the vertical quadrangular decision wait

$$\dot{\Omega}_q(\langle d_1, d_0 \rangle, \langle \mathbf{Zl}, q_8(d_{00}, \sum d_1) x_0 \rangle, (\lambda i. q_8(d_{00}, d_{1i}) y_i)^* \iota_{|d_1|})$$

with dimensions $\sum d_1$ -by- d_{00} , which can be rotated to a horizontal form

$$q_8(\sum d_1, d_{00}) \dot{\Omega}_q(\langle d_1, d_0 \rangle, \langle \mathbf{Zl}, q_8(d_{00}, \sum d_1) x_0 \rangle, (\lambda i. q_8(d_{00}, d_{1i}) y_i)^* \iota_{|d_1|})$$

at no cost. Let this result be denoted $((q_9 q_8) \dot{\Omega}_q)(d, x, y)$ in terms of a function

$$q_9 : ((\mathbb{N} \times \mathbb{N}) \rightarrow (\mathbb{H} \rightarrow \mathbb{H})) \rightarrow (((\mathbb{N}^{*2} \times \mathbb{H}^2 \times \mathbb{H}^*) \rightarrow \mathbb{H}) \rightarrow ((\mathbb{N}^{*2} \times \mathbb{H}^2 \times \mathbb{H}^*) \rightarrow \mathbb{H}))$$

defined by

$$q_9 = \lambda q. \lambda f. \lambda(d, x, y). q(\sum d_1, d_{00}) f(\langle d_1, d_0 \rangle, \langle \mathbf{Zl}, q(d_{00}, \sum d_1) x_0 \rangle, (\lambda i. q(d_{00}, d_{1i}) y_i)^* \iota_{|d_1|}).$$

Covering all cases

Now that we can cope equally well with either $|d_0| = 1$ or $|d_1| = 1$ in a quadrangular decision wait specification, there is a choice between $\dot{\Omega}_q(d, x, y)$ and $((q_9 q_8) \dot{\Omega}_q)(d, x, y)$ for the oddball case of both $|d_0|$ and $|d_1|$ equal to 1. For valid parameters, this condition would imply $|y| = |d_0||d_1| = 1$ with no need for either of x_0 or x_1 in the construction, nor for any completion detecting bus network, and hence a result equivalent to y_0 . Perhaps handling it specially is appropriate even if only a *petaQ* would try to decompose a decision wait into just one part.² Our generalized quadrangular decision wait combining form Ω_q (without the dots) is therefore defined as follows.



$$\Omega_q(d, x, y) = \begin{cases} y_0 & \text{if } |d_0||d_1| = 1 \\ \dot{\Omega}_q(d, x, y) & \text{if } |d_0| > 1 \wedge |d_1| = 1 \\ ((q_9 q_8) \dot{\Omega}_q)(d, x, y) & \text{if } |d_0| = 1 \wedge |d_1| > 1 \\ \dot{\Omega}_q(d, x, y) & \text{otherwise} \end{cases} \quad (10.18)$$

10.3.4 A revised planar decision wait generating function

A decision wait generating function generalizing the one defined by Equation 10.12 to incorporate quadrangular decision waits advantageously could satisfy some sort of a recurrence reducing to Equation 10.12 for dimensions below some low threshold, but not too low. For example, there is

²Klingon word for “idiot”, as in “A doctor who treats himself has a *petaQ* for a patient.”

probably no good reason for 1-by-1 quadrangular decision wait as alluded above, nor even for a 1-by-2, 2-by-1, or 2-by-2, because the routing stages would outweigh the internal array.

Beyond these constraints there is considerable scope for variation. Should any decision wait with either dimension greater than two be decomposed, and if so, into parts of what size, or are cascading forms best for all but the very largest? These questions motivate further investigation in [Appendix C](#). Suffice it to say that informed answers are elusive, but facile and probably wrong answers are readily at hand. For example, a strategy based on the premise that asymptotically logarithmic latency is always preferable, that it is achievable only by quadrangular decomposition, that at least one but no more than two subdivisions along each dimension are ever justified, and that they should be as balanced as possible, implies that an n -by- m decision wait should be decomposed as $\Omega_q(d, x, y)$ with $d = \mathcal{U}_q\langle n, m \rangle$ for a function $\mathcal{U}_q : \mathbb{N}^2 \rightarrow \mathbb{N}^{*2}$ given by

$$\mathcal{U}_q = (\lambda a. \langle \langle a \rangle, \langle \lfloor a/2 \rfloor, \lfloor a/2 \rfloor \rangle \rangle_{\delta_{\{1,2\}} - \{a\}})^*$$

and appropriate choices of x and y to match. Fixing \mathcal{U}_q as shown allows for a revised decision wait generating function satisfying the recurrence

$$\text{DW}(n, m) = \begin{cases} \Omega_p(n, m) & \text{if } \mathcal{U}_q\langle n, m \rangle \in (\mathbb{N}^1)^2 \\ (\lambda d. \Omega_q(d, \text{DW}^* f d, \text{DW}^* b d)) \mathcal{U}_q\langle n, m \rangle & \text{otherwise} \end{cases} \quad (10.19)$$

where the function $f : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$ extracts the correct front end routing stage dimensions from d

$$f = \lambda d. \langle \langle \sum d_0, |d_1| \rangle, \langle |d_0|, \sum d_1 \rangle \rangle$$

and $b : \mathbb{N}^{*2} \rightarrow (\mathbb{N} \times \mathbb{N})^*$ extracts the necessary back end array dimensions (*cf.* [Equation 10.15](#)).

$$b = \lambda d. \flat (\lambda r. (\lambda c. (r, c))^* d_1)^* d_0$$

On the other hand, maybe quadrangular decision waits are over the top and cascading decision waits should be used for everything. In that case, the choice of

$$\mathcal{U}_q\langle n, m \rangle = \langle \langle n \rangle, \langle m \rangle \rangle$$

indicates the cascading form unconditionally in [Equation 10.19](#), making it equivalent to [Equation 10.12](#).

A third alternative is to keep an open mind about how decision waits should be decomposed pending further analysis. This outlook calls for a concept of \mathcal{U}_q not as something set in stone, but as just one member chosen preferably by rational criteria from a broad class of functions $f : \mathbb{N}^2 \rightarrow \mathbb{N}^{*2}$ satisfying

$$\forall s \in \mathbb{N}^2. s = \sum^* f s$$

many of which effect other decompositions than the two extremes noted above in the context of [Equation 10.19](#). Such a function might also be called a **local quadrangular decomposition strategy** hereafter.

With regard to terminology, this type of decomposition strategy is called local rather than global because it provides no means for the decomposition to depend on how the decision wait being decomposed is used within some larger decision wait. [Figure 10.13](#) illustrates a situation where this constraint might be an issue. Suppose a strategy is sought that minimizes the average latency over all inputs. Each of the top i row inputs in the figure depends on a critical path passing through the

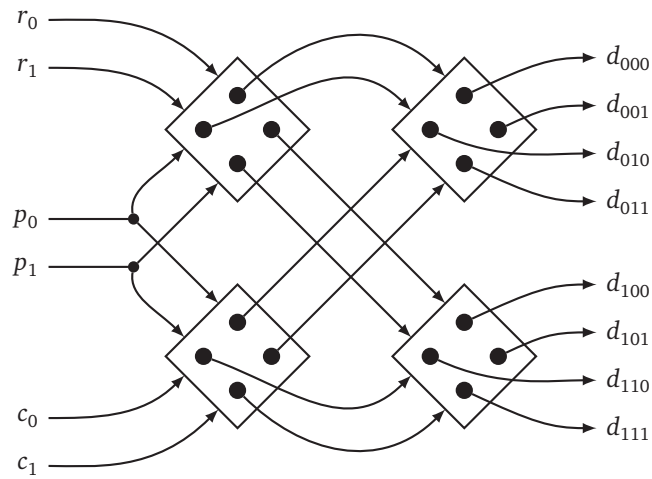


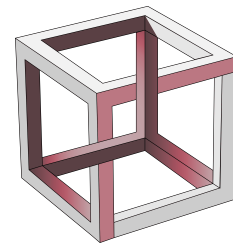
Figure 10.15: A 2-by-2-by-2 decision wait has two planes, two rows, and two columns.

top row of the lower decision wait. Maybe these rows outnumber the lower j rows. If critical paths through different rows are not necessarily identical, could it ever happen that the average latency over all $i + j$ rows would be improved by a strategy that opts for a minimal latency through the top row of the lower decision wait at a cost of increasing the overall average for the lower decision wait? If so, then a strategy that always minimizes the averages locally might not always minimize them globally.

In other words, could this rabbit hole get any deeper? Global decomposition strategies are conceivable, but not without a few other things to cover first.

10.4 Multidimensional decision waits

Deferring further discussion of decomposition strategies temporarily, we have an opportunity to explore the design space in literally a different dimension. The decision waits described up to this point, whether cascading or quadrangular, are all *planar*. Planar decision waits are two dimensional, but higher dimensions are possible. An n -dimensional decision wait has input terminals partitioned into n sets and accepts a signal on exactly one input from each set concurrently before emitting exactly one output signal. The number of output terminals is equal to the product of the cardinalities of the sets of input terminals in the partition. For example, a 2-by-2-by-2 decision wait would be three dimensional, with two input terminals in each dimension, and eight outputs. A design for a 2-by-2-by-2 decision wait due to [294] is shown in Figure 10.15.



Two salient questions about multidimensional decision waits are whether they can be designed in any uniform way for arbitrary dimensions, and whether they are of any use. One use for a 2-by-2-by-2 decision wait would be as a three bit dual rail decoder. (See the footnote on page 45.) As such,

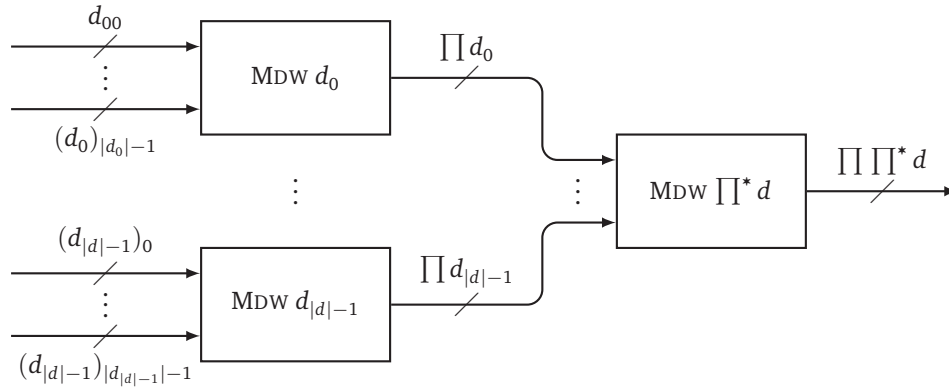


Figure 10.16: a dendriform multidimensional decision wait with dimensions $d \in \mathbb{N}^{**}$

it allows for the communication of any of eight possible code words over a bus with only six lines, which may be a worthwhile tradeoff if bus lines are expensive. An n -bit dual rail decoder would require an n -dimensional decision wait, and would cut down the number of bus lines from 2^n to $2n$. There are many other delay insensitive codes and decoders for them using multidimensional decision waits discussed in Chapter 13.

The other question posed above occupies the rest of this section. There are at least two reasonably straightforward ways to design arbitrary multidimensional decision waits in general, each subject to many variations, but both easier than the quadrangular decomposition for planar decision waits. One designated as the **dendriform** decision wait follows in Section 10.4.1, and the other designated as the **crossbar** decision wait follows in Section 10.4.2, the latter being a direct generalization of the idea behind Figure 10.15. Their development proceeds similarly to the quadrangular decision wait in that a combining form and a complementary decomposition strategy determine a decision wait generating function defined by a recurrence.

10.4.1 Dendriform

The dendriform decision wait is the simpler of the two. The idea is to build a tree of lower dimensional decision waits as shown in Figure 10.16, in which all of the outputs from each leaf node connect to the inputs along a single dimension of the root. For example, a couple of 2-by-2 planar decision waits could be used as leaves connected in this way to a 4-by-4 planar decision wait used as the root to create a 2-by-2-by-2-by-2 decision wait. All four outputs from the first leaf would be connected to the row inputs of the root, and all four outputs from the second leaf would be connected to the column inputs of the root.

In general, this construction allows any number of leaves greater than one, each having any positive number of positive dimensions, provided the root has compatible dimensions. Even a one dimensional decision wait is allowed, which is just a bus. Compatibility in this context depends on the fact that an $|s|$ -dimensional decision wait with dimensions $s \in \mathbb{N}^*$ has $\sum s$ inputs and $\prod s$ outputs. Its dendriform decomposition into $|d|$ leaves whose i -th leaf has dimensions $d_i \in \mathbb{N}^*$ for some $d \in \mathbb{N}^{**}$ satisfying $\flat d = s$ requires a $|d|$ -dimensional root with dimensions $\prod^* d \in \mathbb{N}^*$,

hence $\sum \prod^* d$ inputs and $\prod \prod^* d$ outputs. Best of all, there is no need to bother with permutation networks because all of the inputs and outputs fall naturally into the right order.

Combining form

Similarly to a quadrangular decision wait, the dendriform decision wait is appropriate to describe by a combining form Ω_d parameterized by the dimensions and taking its parts to the whole. For dimensions $s \in \mathbb{N}^*$, these parameters would be a list $d \in \mathbb{N}^{**}$ meeting the conditions above, a list $x \in \mathbb{H}^{|d|}$ of multidimensional decision waits for the leaves wherein each x_i has dimensions d_i , and a multidimensional decision wait $y \in \mathbb{H}$ for the root with dimensions $\prod^* d$. The combining form

$$\Omega_d : \mathbb{N}^{**} \times \mathbb{H}^* \times \mathbb{H} \rightarrow \mathbb{H}$$

could hardly be simpler.

$$\Omega_d(d, x, y) = \mathbf{C}_{\sum \prod^* d} \langle (f \mathbf{R}) x, y \rangle \quad (10.20)$$

Local decomposition strategy

To determine a particular multidimensional decision wait generating function $\text{MDW} : \mathbb{N}^* \rightarrow \mathbb{H}$ based on Ω_d , a **local dendriform decomposition strategy** $\mathcal{U}_d : \mathbb{N}^* \rightarrow \mathbb{N}^{**}$ would have to fix all of the its discretionary features. At a minimum, it should satisfy

- $s = \flat \mathcal{U}_d s$
- $|\mathcal{U}_d s| > 1$
- $\epsilon \notin \mathcal{R}(\mathcal{U}_d s)$

for all lists $s \in (\mathbb{N} - \{0\})^* - \bigcup_{i=0}^2 \mathbb{N}^i$. For example, a strategy opting exclusively for planar nodes throughout the tree (and therefore admitting only binary trees) would be

$$\mathcal{U}_d(s) = \langle s \upharpoonright \lfloor s/2 \rfloor, s \ll \lfloor s/2 \rfloor \rangle.$$

Generating function

For any valid choice of \mathcal{U}_d , we have a corresponding multidimensional decision wait generating function

$$\text{MDW}(s) = \begin{cases} \langle \Gamma^{s_0}, \text{DW}(s_0, s_1) \rangle_{\delta_2^{|s|}} & \text{if } |s| \leq 2 \\ (\lambda d. \Omega_d(d, \text{MDW}^* d, \text{MDW} \prod^* d)) \mathcal{U}_d s & \text{otherwise} \end{cases} \quad (10.21)$$

assuming a fixed planar decision wait generating function $\text{DW} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$. Regrettably, a dendriform decomposition strategy \mathcal{U}_d is even more localized than \mathcal{U}_q in the sense that it can not affect the choice of planar decision waits supporting it, further incentivizing a more comprehensive solution.

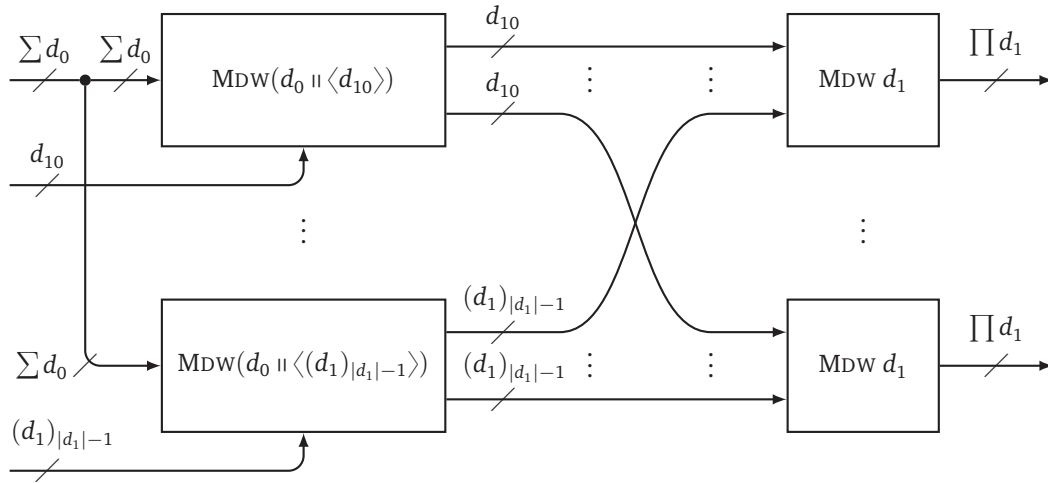


Figure 10.17: a crossbar multidimensional decision wait combining form with dimensions $d \in \mathbb{N}^{*2}$

10.4.2 Crossbar

Staying focused on the task at hand, we have another way of designing multidimensional decision waits to consider. The design in Figure 10.15 is not a special snowflake but an instance of a class of decompositions consisting of two arrays of lower dimensional decision waits with each member of the first array connected to every member of the second, as shown in Figure 10.17. For an $|s|$ -dimensional decision wait, this construction entails a split of the dimensions s into exactly two non-empty sublists d_0 and d_1 with $d_0 \parallel d_1 = s$.

- The back end consists of $\prod d_0$ decision waits each with dimensions d_1 , while the front end consists of $|d_1|$ decision waits each having $|d_0| + 1$ dimensions.
- The front end decision waits differ only in their last dimension, having one representative for each term of d_1 , while their other dimensions are identically d_0 .
- Each decision wait in the back end receives the inputs associated with its i -th dimension on a bus of width d_{1i} coming from the i -th decision wait in the front end.
- Each decision wait in the front end sees the same signals for all inputs in its first d_0 dimensions, but receives a signal specific to itself along its last dimension.

The idea is that the first $|d_0|$ input buses select the same output bus on every front end decision wait (of which there are $\prod d_0$ on each), thereby causing whatever signal each of them receives along its last dimension to be routed to exactly the same back end decision wait and no other. Having received a full complement of $|d_1|$ input signals, only one back end decision wait emits an output.

Combining form

A crossbar decision wait $\Omega_c(d, x, y) \in \mathbb{H}$ with dimensions $\flat d \in \mathbb{N}^*$ is specified by a combining form

$$\Omega_c : \mathbb{N}^{*2} \times \mathbb{H}^* \times \mathbb{H}^* \rightarrow \mathbb{H}$$

a list of two lists of dimensions $d \in ((\mathbb{N} - \{0\})^* - \{\epsilon\})^2$, a list $x \in \mathbb{H}^{|d_1|}$ of front end stage decision waits, and a list $y \in \mathbb{H}^{\prod d_0}$ of back end stage decision waits, where each front end term x_i has dimensions $d_0 \parallel \langle d_{1i} \rangle$ and each back end term y_j has identical dimensions d_1 . In front of the front end, it has an array of $\sum d_0$ FORK networks each having $|d_1|$ outputs

$$(\text{FORK}|d_1|)^{\sum d_0}$$

to connect each of the $\sum d_0$ inputs from the first $|d_0|$ input buses to the same position on each of the front end decision waits in x . A further $\sum d_1$ inputs are bundled into a block $\mathbb{I}^{\sum d_1}$ of $|d_1|$ buses such that the i -th bus is destined for the last d_{1i} inputs on x_i . A rearrangement of the FORK outputs

$$(\text{FORK}|d_1|)^{\sum d_0} \times \iota_{|d_1|\sum d_0} \times \sum d_0$$

provides for the i -th of the first $|d_1|$ consecutive groups of $\sum d_0$ outputs to be associated with the i -th front end decision wait x_i just as the i -th of the next $|d_1|$ buses is already, in an expression

$$\mathbf{R}((\text{FORK}|d_1|)^{\sum d_0} \times \iota_{|d_1|\sum d_0} \times \sum d_0, \mathbb{I}^{\sum d_1})$$

making it a bit easier to find a permutation $k_0 d \in \mathbb{N}^*$ whereby the whole front end can be expressed

$$\mathbf{R}((\text{FORK}|d_1|)^{\sum d_0} \times \iota_{|d_1|\sum d_0} \times \sum d_0, \mathbb{I}^{\sum d_1}) \xrightarrow{k_0 d} (\mathcal{F} \mathbf{R}) x$$

for some $k_0 : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$.

Among the first $|d_1|$ buses from here, which each have a width of $\sum d_0$, the permutation $k_0 d$ must route the i -th to a position on $(\mathcal{F} \mathbf{R}) x$ offset by the $(i \sum d_0) + \sum(d_{1 \mid i})$ terminals preceding it

$$\flat (\lambda i. \iota_{\sum d_0}^{(i \sum d_0) + \sum(d_{1 \mid i})})^* \iota_{|d_1|}$$

and it must route the i -th of the next $|d_1|$ buses, which has a width d_{1i} , to a similar position offset by the additional $\sum d_0$ terminals preceding it

$$\flat (\lambda i. \iota_{d_{1i}}^{((i+1) \sum d_0) + \sum(d_{1 \mid i})})^* \iota_{|d_1|}$$

suggesting the result

$$k_0 = \lambda d. \flat(((\lambda i. \iota_{\sum d_0}^{(i \sum d_0) + \sum(d_{1 \mid i})})^* \iota_{|d_1|}) \parallel ((\lambda i. \iota_{d_{1i}}^{((i+1) \sum d_0) + \sum(d_{1 \mid i})})^* \iota_{|d_1|})).$$

Connecting the front end to the back end $(\mathcal{F} \mathbf{R}) y$ in terms of another permutation $k_1 d \in \mathbb{N}^*$ suggests an expression

$$\Omega_c(d, x, y) = \mathbf{R}((\text{FORK}|d_1|)^{\sum d_0} \times \iota_{|d_1|\sum d_0} \times \sum d_0, \mathbb{I}^{\sum d_1}) \xrightarrow{k_0 d} (\mathcal{F} \mathbf{R}) x \xrightarrow{k_1 d} (\mathcal{F} \mathbf{R}) y \quad (10.22)$$

for the whole combining form, subject only to finding a suitable $k_1 : \mathbb{N}^{*2} \rightarrow \mathbb{N}^*$. This permutation can be inferred from the fate of the j -th output bus on the i -th front end decision wait x_i , which has

a width of d_{1i} and is one of $\prod d_0$ output buses from x_i . It must reach the j -th back end decision wait y_j , whose j predecessors have a combined input arity of $j \sum d_1$. Relative to y_j , it must follow the i input buses preceding it, whose combined width is $\sum (d_1 + i)$. Hence it accounts for a sublist

$$t_{d_{1i}}^{(j \sum d_1) + \sum (d_1 + i)}$$

of the permutation $k_1 d$. Combined with the rest of the $\prod d_0$ buses from x_i , it accounts for a sublist

$$b (\lambda j. t_{d_{1i}}^{(j \sum d_1) + \sum (d_1 + i)})^* t_{\prod d_0}$$

whose combination with the sublists due to the rest of the $|d_1|$ terms of x makes up all of $k_1 d$ based on a definition

$$k_1 = \lambda d. b^2 (\lambda i. (\lambda j. t_{d_{1i}}^{(j \sum d_1) + \sum (d_1 + i)})^* t_{\prod d_0})^* t_{|d_1|}$$

thus completing the specification of Ω_c in Equation 10.22.

Generating function

A **local crossbar decomposition strategy** $\mathcal{U}_c : \mathbb{N}^* \rightarrow \mathbb{N}^{*2}$ must meet the same conditions as a local dendriform decomposition strategy mentioned on page 300 in addition to

$$\forall s \in \mathbb{N}^* - \bigcup_{i=0}^2 \mathbb{N}^i. |\mathcal{U}_c s| = 2$$

obviously, and is subject to the same limitations mentioned on page 300. Any valid local crossbar decomposition strategy determines a multidimensional decision wait generating function satisfying this recurrence.

$$MDW(s) = \begin{cases} \langle \mathbb{F}^0, DW(s_0, s_1) \rangle_{\delta_2^{|s|}} & \text{if } |s| \leq 2 \\ (\lambda d. \Omega_c(d, (\lambda i. MDW(d_0 \parallel \langle i \rangle))^* d_1, (MDW d_1) \prod d_0)) \mathcal{U}_c s & \text{otherwise} \end{cases} \quad (10.23)$$

For example, one that always opts for planar front ends would be

$$\mathcal{U}_c(s) = \langle \langle s_0 \rangle, s \ll 1 \rangle.$$

With Equation 10.21 and Equation 10.23, we now have two competing definitions of a multidimensional decision wait generating function $MDW : \mathbb{N}^* \rightarrow \mathbb{H}$ with no indication of which to prefer and no smooth way of integrating one with the other. There can be little hope for any improvement in this state of affairs without further investigation, but fortunately the chapter is not over yet.

10.5 Decision wait transformations

Although decision wait rotations have proved helpful for expressing horizontal quadrangular forms in Section 10.3.3, is a deep dive into general rotations and permutations for decision waits of arbitrary dimensions really justified? While they are not useful for expressing decision waits of yet more general shapes and sizes (because there are none), they may be of interest for performance reasons. For example, if some rows in a decision wait built according to a particular local decomposition strategy were slower than others due to some unavoidable asymmetry, then a cascading sequence of identically oriented instances could exhibit a cumulative imbalance in latency. This effect might be correctable just by reversing the rows on alternating instances. Such transformations are worth having at our disposal and are readily amenable to analysis.



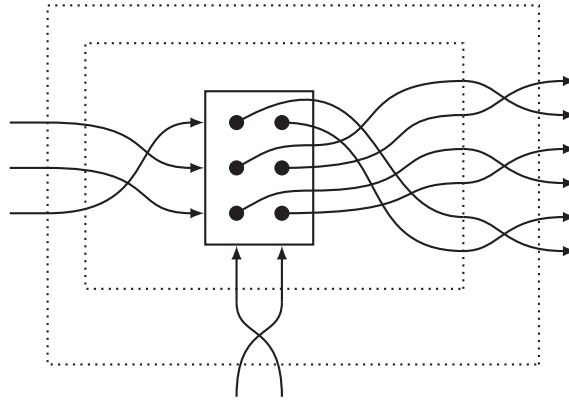


Figure 10.18: To maintain behavioral equivalence, the outputs from each row have to be permuted inversely to the column input permutation, and the buses from the rows have to be permuted inversely to the row input permutation (inset).

10.5.1 Permuting along the axes

The transformation suggested above requires both an input and an output permutation network to preserve behavioral equivalence. A decision wait could have not just its rows but its inputs in every dimension reversed, and not just reversed but permuted arbitrarily. A complete description of any such transformation to an $|s|$ -dimensional decision wait with dimensions $s \in \mathbb{N}^*$ could be specified by a list of permutations $p \in \mathbb{N}^{*|s|}$ of length $|s|$ satisfying $|p_i| = s_i$ for all $0 \leq i < |s|$. For example, the two dimensional decision wait in Figure 10.18 with dimensions $s = \langle 3, 2 \rangle$ is subject to both a row input permutation $p_0 = \langle 1, 2, 0 \rangle$ and a column input permutation $p_1 = \langle 1, 0 \rangle$. The result is still a decision wait with the same dimensions because the outputs are permuted in a way that compensates for the input permutations.

To describe this transformation formally, we envision an $|s|$ -dimensional decision wait $x \in \mathbb{H}$ having a behavioral equivalent $(\phi_0 p) x \in \mathbb{H}$ permuted according to a list of permutations $p \in \mathbb{N}^{*|s|}$ meeting the conditions above in terms of a function $\phi_0 p : \mathbb{H} \rightarrow \mathbb{H}$ determined by second order function $\phi_0 : \mathbb{N}^{**} \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$. The result $(\phi_0 p) x$ should be something like x with a wrapper of input and output permutation networks around it depending on p .

The input permutation network is the easy part, requiring only an array of individual permutation networks $a \times |^{|a|}$ in parallel for each permutation $a \in \mathcal{R}(p)$

$$(\lambda a. a \times |^{|a|})^* p$$

and hence a partial result

$$\mathbf{C}_{|p|} \langle (\mathcal{F} \mathbf{R}) (\lambda a. a \times |^{|a|})^* p, x \rangle.$$

The output permutation is best to build incrementally by generalizing from Figure 10.18. The outputs from any $|s|$ -dimensional decision wait naturally form a number of consecutive buses each having a width $s_{|s|-1}$, the last dimension. Each of these buses must be permuted inversely to the last permutation $p_{|p|-1}$, so we can expect a pattern of some number of replicated copies of $p_{|p|-1}$ in the network. On a higher level, the buses themselves are permuted relative to one another

inversely to the penultimate permutation in p , and this pattern repeats on yet another level for each dimension. At the highest level, we would make s_0 copies of a network described by some permutation t , and then permute the copies by p_0 relative to one another. This network would be described by a concatenated sequence of lists of the form

$$(\lambda j. i|t| + j)^* t$$

with a different value of i for each term in the permutation p_0 .

$$b (\lambda i. (\lambda j. i|t| + j)^* t)^* p_0$$

The whole output permutation is therefore given by a fold over p

$$(\mathcal{F}_{\langle 0 \rangle} \lambda(a, t). b (\lambda i. (\lambda j. i|t| + j)^* t)^* a) p$$

and the whole definition for the transformation by

$$\phi_0 = \lambda p. \lambda x. \mathbf{C}_{|b|p|} \langle (\mathcal{F} \mathbf{R}) (\lambda a. a \times |a|)^* p, x \rangle \times (\mathcal{F}_{\langle 0 \rangle} \lambda(a, t). b (\lambda i. (\lambda j. i|t| + j)^* t)^* a) p. \quad (10.24)$$

10.5.2 Permuting the axes

The other kind of decision wait transformation not covered by the foregoing is a rotation generalizing Equation 10.17, which changes an n -by- m decision wait to an m -by- n decision wait. This problem escalates with higher dimensions. For example, an l -by- n -by- m decision wait could be rotated into one with dimensions l -by- m -by- n , m -by- l -by- n , m -by- n -by- l , n -by- l -by- m , or n -by- m -by- l . In general, an $|s|$ -dimensional decision wait with dimensions $s \in \mathbb{N}^*$ has $|s|!$ possible orientations, one for each permutation of the dimensions, so even a rotation is really a permutation.



The idea of a rotational transformation to a decision wait $x \in \mathbb{H}$ with dimensions $s \in \mathbb{N}^*$ is captured by a function $\phi_1 : \mathbb{N}^* \times \mathbb{N}^* \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ such that $\phi_1(p, s) x \in \mathbb{H}$ is a decision wait derived from x with dimensions $s \circ p = (\lambda r. s_r)^* p$, where $p \in \mathbb{N}^{|s|}$ is a permutation of length $|s|$. The resulting decision wait $\phi_1(p, s) x \in \mathbb{H}$ exposes $\sum s$ input terminals on a permutation network of $|s| = |p|$ consecutive buses wherein the i -th bus has a width of s_{p_i} and connects to x starting at the terminal numbered $\sum (s \upharpoonright p_i)$, implying an input permutation network described by

$$b (\lambda r. t_{s_r}^{\sum (s \upharpoonright r)})^* p \in \mathbb{N}^{\sum s}.$$

To get a handle on the output permutation, we can regard each output from the $|s|$ -dimensional decision wait x before the transformation as a point with coordinates

$$b = \langle b_0 \dots b_{|s|-1} \rangle \in \mathbb{N}^{|s|}$$

in an $|s|$ -dimensional lattice, with each coordinate b_i ranging from 0 to $s_i - 1$. The output with coordinates b is the one that emits a signal whenever the b_i -th of s_i possible inputs is received along i -th axis for all $0 \leq i < |s|$. The list $\bar{t}_s \in (\mathbb{N}^{|s|})^{\prod s}$ contains the coordinates of all $\prod s$ outputs from x in lexicographic or “row major” order, which is the order of their output terminals. After the transformation, the dimensions $s \circ p$ of the result $\phi_1(p, s) x$ mean there should be an output with coordinates $a = b \circ p$ associated with it for each point b associated with x . To maintain the row major order convention, the position of the output terminal with coordinates a relative to the other

output terminals on the result should be the position of a in the list $\bar{t}_{s \circ p}$. Because a corresponds to b , the output permutation network should run a wire to this terminal from the terminal with coordinates $b = a \circ p^{-1}$ on x , which comes from the position $(\bar{t}_s)^{-1}(a \circ p^{-1})$ of the output bus from x , implying an output permutation

$$(\lambda a. (\bar{t}_s)^{-1}(a \circ p^{-1}))^* \bar{t}_{s \circ p}$$

and the following definition for ϕ_1 overall.

$$\phi_1 = \lambda(p, s). \lambda x. \flat (\lambda r. \iota_{s_r}^{\sum(s_i r)})^* p \times x \times (\lambda a. (\bar{t}_s)^{-1}(a \circ p^{-1}))^* \bar{t}_{s \circ p}$$

10.5.3 Permuting and rotating

A general transformation that integrates both of those above is convenient to encapsulate in a single function $\phi : \mathbb{N}^{**} \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ parameterized by the list of permutations $p \in \mathbb{N}^{**}$.

$$\phi(p) = (\lambda p'. \phi_1(p_0, (\lambda a. |a|)^* p') \circ \phi_0 p') (p \ll 1) \quad (10.25)$$

The intent is that $\phi p : \mathbb{H} \rightarrow \mathbb{H}$ permutes a decision wait x having dimensions s to one $\phi_0(p \ll 1) x$ still having dimensions s , and rotates that to a final decision wait $\phi_1(p_0, s) \phi_0(p \ll 1) x$ having dimensions $s \circ p_0$. Dimensions s satisfying $|s| = |p_0| = |p| - 1$ and $s_i = |p_{i+1}|$ are inferred from p .

10.6 Optimized decision waits

As noted previously, the best way to decompose a decision wait might depend on its context. Local decomposition strategies are not always accommodating of this possibility, but there is an alternative. A **global decomposition** takes the form of an ordered tree (Section 10.1), balanced or unbalanced, binary, ternary, or m -ary, and as flat or as deeply nested as we please, with every decision wait used as a building block in the construction described explicitly by its own particular subtree. In this way, decision waits with similar dimensions but different decompositions are readily expressible as such by different subtrees within the same global decomposition (preferably to some worthwhile end). A **global decomposition strategy** is a function that takes any desired dimensions to a global decomposition describing a decision wait with those dimensions.

The advantage of working with global decompositions is that they map the design space with far better coverage than local decompositions can. An explicit concrete model of global decompositions as ordered trees establishes a framework for enumerating many possible designs algorithmically or heuristically in search of an optimum with respect to any given metric, hence the title of this section. Reducing the problem to these terms could perhaps leverage the techniques of Monte Carlo Tree Search (MCTS) normally used for finding optimum game playing strategies (surveyed in [39]), particularly as they pertain to single player games [239], based on players' utility functions derived from cost or performance metrics.

However, the rest of this section sticks to the subject at hand. Just as a local decomposition strategy such as \mathcal{U}_q relies on an associated combining form like Ω_q to be useful in a recurrence like Equation 10.19, global decomposition strategies also entail corresponding combining forms. Both are described for each of the quadrangular, dendriform, and crossbar decompositions, along with one that ultimately unifies all of them, in Section 10.6.2 through Section 10.6.5 respectively after a few formalities are out of the way in Section 10.6.1.



10.6.1 Global decompositions

A global decomposition for a decision wait is defined formally as an ordered tree in $\mathcal{O}(\mathbb{N}^{**} \times \mathbb{N}^{**})$ wherein each node $(p, d) \in \mathbb{N}^{**} \times \mathbb{N}^{**}$ contains a list of permutations $p \in \mathbb{N}^{**}$ and a list of lists of dimensions $d \in \mathbb{N}^{**}$ with the following intended interpretations.

- To generate the decision wait described by a non-terminal subtree $((p, d), v)$ whose root node is (p, d) , the list of lists of dimensions d is used as the first parameter to one of the combining forms Ω_q , Ω_d , or Ω_c (Equation 10.18, Equation 10.20 or Equation 10.22), with other parameters obtained from the decision waits generated from the subtrees v .
- The list of permutations p parameterizes the function ϕ defined by Equation 10.25 in a function $\phi p : \mathbb{H} \rightarrow \mathbb{H}$ to be applied to the decision wait constructed by the combining form above, implying dimensions of $(\lambda r. |p_{r+1}|)^* p_0 \in \mathbb{N}^*$ for the result if the dimensions of d and the subtrees are consistent with it.
- To generate the decision wait described by a terminal subtree $((p, d), \epsilon)$, a similar interpretation applies to the list of permutations p , but $(\sum d_0, \sum d_1)$ parameterizes Ω_p (Equation 10.11) instead of d parameterizing Ω_q , Ω_d , or Ω_c .

To start filling some of the gaps in this description, let the universe of decision wait decompositions be constrained to a set $\mathbb{S} \subset \mathcal{O}(\mathbb{N}^{**} \times \mathbb{N}^{**})$ defined as

$$\mathbb{S} = \{t \in \mathcal{O}((\mathbb{N}^* - \{\epsilon\})^* \times ((\mathbb{N} - \{0\})^* - \{\epsilon\})^*) \mid (\Lambda \lambda((p, d), v). \prod (v \parallel (\lambda a. \delta_{\mathcal{D}(a)}^{\mathcal{R}(a)}))^* p) t = 1\}$$

which is to say that p can contain only non-empty permutations (lists with domains equal to their ranges), and d can contain only non-empty lists of non-zero dimensions. (See Equation 10.1 for a reminder about the tree folding combinator notation.)

Even then, not all members $t \in \mathbb{S}$ are meaningful as global decompositions because this constraint does nothing to reconcile dimensional inconsistencies among p , d , and v . A global decomposition with a root (p, d) and subtrees v describes a decision wait with the dimensions $(\lambda r. |p_{r+1}|)^* p_0$ noted above only if d and v also meet certain conditions. These conditions may depend on the relevant combining form Ω_p , Ω_q , Ω_d or Ω_c . One way of indicating them verifiably is by a function

$$\psi \Delta : \mathbb{S} \rightarrow \mathbb{N}^*$$

taking a decomposition $t = ((p, d), v) \in \mathbb{S}$ to a list of dimensions $(\psi \Delta) t$ equal to $(\lambda r. |p_{r+1}|)^* p_0$ if t meets the conditions, but to an empty list ϵ otherwise. Then the function $\psi \Delta$ can be parameterized by a function

$$\Delta : \mathbb{N}^{**} \times \mathbb{N}^{**} \times \mathbb{N}^{**} \rightarrow \{0, 1\}$$

such that Δ takes p , d , and the list $s \in \mathbb{N}^{**}$ of lists of dimensions similarly obtained from the subtrees v to a non-zero value $\Delta(p, d, s) \in \{0, 1\}$ only if d , v , and s are as they should be for the given p and the stipulated combining form. This convention implies a second order function

$$\psi : ((\mathbb{N}^{**} \times \mathbb{N}^{**} \times \mathbb{N}^{**}) \rightarrow \{0, 1\}) \rightarrow (\mathbb{S} \rightarrow \mathbb{N}^*)$$

given by

$$\psi = \lambda f. \Lambda \lambda((p, d), s). \langle \lambda i. \langle \epsilon, (\lambda r. |p_{r+1}|)^* p_0 \rangle_i \delta_{|p|}^{|p_0|+1} f(p, d, s), \epsilon \rangle_{\delta_\epsilon} \quad (10.26)$$

taking the opportunity to require the condition $|p| = |p_0| + 1$ regardless, which simplifies the tasks ahead insofar as Δ need not provide for the alternative.

10.6.2 Quadrangular

A global decomposition $t \in \mathbb{S}$ for a planar decision wait describes the cascading form or a bus when t is terminal and the quadrangular form otherwise. A terminal t implies dimensions $(\psi \Delta_q^t) t \in \mathbb{N}^2$ with respect to Equation 10.26 when t is valid, where Δ_q^t is defined as

$$\Delta_q^t = \lambda(p, d, s). \lambda(p, d, s). (\lambda i. \langle 0, \delta_1^{|d_0|} \delta_{d_0}^{|p_1|} \delta_{d_1}^{|p_2|} \rangle_i) \delta_2^{|d|} \delta_3^{|p|} \delta_s^\epsilon$$

meaning there are three permutations, two unit lists of dimensions respectively matching the latter permutation lengths, and no subtrees. These conditions are in keeping with the specification for a cascading decision wait $\Omega_p(\sum d_0, \sum d_1) \in \mathbb{H}$ stated above.

The conditions for non-terminal trees $t \in \mathbb{S}$ differ in that the dimensions of the first two subtrees must be those of the front end routing stages x in a quadrangular decomposition, and the dimensions of the rest of the subtrees must be those of the internal decision waits y in row major order as shown in Figure 10.9. A related function Δ_q^n captures these conditions

$$\Delta_q^n = \lambda(p, d, s). \langle 0, (\lambda k. \delta_{\sum d_0}^{|p_1|} \delta_{\sum d_1}^{|p_2|} \delta_k^s) \langle \sum d_0, |d_1| \rangle : \langle |d_0|, \sum d_1 \rangle : b(\lambda i. (\lambda j. \langle i, j \rangle)^* d_1)^* d_0 \rangle_{\delta_3^{|p|} \delta_2^{|d|}}$$

and one that covers both the terminal and non-terminal cases with respect to Equation 10.26 is given by

$$\Delta_q = \lambda r. (\Delta_q^t r) + (\Delta_q^n r) \quad (10.27)$$

meaning that any decomposition $t \in \mathbb{S}$ with dimensions $(\psi \Delta_q) t \in \mathbb{N}^2$ can be said to describe a planar decision wait in quadrangular or cascading form.

This characterization enables a precise definition for a **global quadrangular decomposition strategy** as any function $\hat{\mathcal{U}}_q : \mathbb{N}^2 \rightarrow \mathbb{S}$ satisfying

$$\forall s \in (\mathbb{N} - \{0\})^2. s = (\psi \Delta_q) \hat{\mathcal{U}}_q s \quad (10.28)$$

which is to say that for two positive dimensions $n, m \in \mathbb{N}$, a global decomposition of an n -by- m decision wait is available as $\hat{\mathcal{U}}_q \langle n, m \rangle$. This decomposition then determines a decision wait

$$\text{DW}(n, m) = \hat{\Omega}_q \hat{\mathcal{U}}_q \langle n, m \rangle \quad (10.29)$$

with the combining form $\hat{\Omega}_q : \mathbb{S} \rightarrow \mathbb{H}$ defined in the obvious way

$$\hat{\Omega}_q = \Lambda \lambda((p, d), v). (\phi_p) \langle \Omega_q(d, v \uparrow 2, v \ll 2), \Omega_p(\sum d_0, \sum d_1) \rangle_{\delta_v^\epsilon} \quad (10.30)$$

based in Equation 10.11, Equation 10.18, and Equation 10.25.

Equation 10.29 could be taken as our most sophisticated candidate yet for a definition of a decision wait generating function if only some global decomposition strategy $\hat{\mathcal{U}}_q$ were known. It is at least as capable as Equation 10.19 because the global decomposition strategy could always be chosen as

$$\hat{\mathcal{U}}_q(s) = (\lambda f. f \mathcal{U}_q s) \lambda d. \begin{cases} (\langle \langle t_2, t_{\sum d_0}, t_{\sum d_1} \rangle, d \rangle, \epsilon) & \text{if } d \in (\mathbb{N}^1)^2 \\ (\langle \langle t_2, t_{\sum d_0}, t_{\sum d_1} \rangle, d \rangle, b(\lambda i. (\lambda j. \hat{\mathcal{U}}_q \langle i, j \rangle)^* d_1)^* d_0) & \text{otherwise} \end{cases}$$

for any local decomposition strategy \mathcal{U}_q (i.e., with no rotations, no permutations, and no context dependence) to yield results identical to those of Equation 10.19, including $\mathcal{U}_q \langle n, m \rangle = \langle \langle n \rangle, \langle m \rangle \rangle$ for exclusively cascading results. However, the strength of this approach would be in choosing

$$\hat{\mathcal{U}}_q = \lambda s. (\lambda(m, t). t) \min\{(\|\hat{\Omega}_q t\|, t) \in \mathbb{R} \times \mathbb{S} \mid (\psi \Delta_q) t = s\}$$

for some real-world real valued metric $\|z\| \in \mathbb{R}$ on circuits $z \in \mathbb{H}$ to optimize the global decomposition strategy by sampling all decompositions t having the desired dimensions s . For example, the metric

$$\|z\| = |\mathcal{T}_{\mathbb{H}\mathbb{L}} z|$$

would select the strategy having the minimum component count as determined by the length of the netlist (Equation 8.23), but this metric by itself is one of the least interesting and is possible to compute in any case directly from the decomposition without generating the circuit (Section C.1).

10.6.3 Dendriform

A way of benefiting in part from a global decomposition strategy for multidimensional decision waits is to restrict attention to a family of trees $u \in \mathbb{S}$ whose non-terminal nodes express local dendriform decompositions and whose terminal nodes express the dimensions of the planar building blocks but not their decompositions. A valid choice of u along with a fixed quadrangular decomposition strategy $\hat{\mathcal{U}}_q$ would then provide a recipe for building a dendriform decision wait. If these restrictions are acceptable, the solution is at hand. If not, there are still Section 10.6.4 and Section 10.6.5.

Global dendriform decompositions

Normally a dendriform decision wait decomposition $u = ((p, d), v) \in \mathbb{S}$ entails a list $d \in \mathbb{N}^{**}$ of any number of lists of dimensions $d_i \in \mathbb{N}^*$, with each being the dimensions of a building block, but at the lowest level the number of building blocks reduces to $|d| = 1$, and the one lowly building block has either $|d_0| = 1$ or $|d_0| = 2$ dimensions. A one dimensional decision wait reduces to a bus \mathbb{I}^{d_0} , while a two dimensional d_{00} -by- d_{01} decision wait is a quadrangular or cascading form $\hat{\Omega}_q \hat{\mathcal{U}}_q d_0$ as usual. Inferring the dimensions d_0 from u in these cases subject also to the validity of u as a decomposition would be a matter of evaluating $(\psi \Delta_d^t) u \in \mathbb{N}^*$ with Δ_d^t defined as

$$\Delta_d^t = \lambda(p, d, s). \delta_1^{|d|} \delta_e^s \langle 0, \prod (\lambda i. \delta_{(b d)_i}^{|p_{i+1}|})^* \iota_{|b d|} \rangle_{\delta_{|p|}^{|b d|+1}} \quad (10.31)$$

which is non-zero only if u is terminal and has permutations p compatible with d . When u is non-terminal, it describes a $|b d|$ -dimensional decision wait built of $|d| > 1$ blocks, of which the i -th has dimensions $d_i = s_i$ matching those of the i -th subtree v_i . An additional subtree $v_{|d|}$ describes a decision wait having dimensions $s_{|d|} = \prod^* d$ conforming to the root block in a dendriform decomposition parameterized by d according to Equation 10.20 if the sanity check $b d = (\psi \Delta_d) u$ holds for a function

$$\Delta_d = \lambda r. (\Delta_d^t r) + (\Delta_d^n r)$$

with the non-terminal cases covered by

$$\Delta_d^n = \lambda(p, d, s). (1 - \delta_1^{|d|}) ((\lambda k. \delta_k^s) (d \parallel \langle \prod^* d \rangle)) \langle 0, \prod (\lambda i. \delta_{(b d)_i}^{|p_{i+1}|})^* \iota_{|b d|} \rangle_{\delta_{|p|}^{|b d|+1}} \quad (10.32)$$

to ensure compatible permutation lengths.

Decomposition strategies

These observations suggest a formal definition for a **global dendriform decomposition strategy** as any function $\hat{\mathcal{U}}_d : \mathbb{N}^* \rightarrow \mathbb{S}$ satisfying

$$\forall s \in (\mathbb{N} - \{0\})^* - \{\epsilon\}. s = (\psi \Delta_d) \hat{\mathcal{U}}_d s$$

(cf. Equation 10.28) and a related combining form $\hat{\Omega}_d \hat{\Upsilon}_q : \mathbb{S} \rightarrow \mathbb{H}$ parameterized by a global quadrangular decomposition strategy $\hat{\Upsilon}_q : \mathbb{N}^2 \rightarrow \mathbb{S}$ with the function $\hat{\Omega}_d : (\mathbb{N}^2 \rightarrow \mathbb{S}) \rightarrow (\mathbb{S} \rightarrow \mathbb{H})$ defined as

$$\hat{\Omega}_d = \lambda f. \Lambda \lambda((p, d), v). (\phi p) \langle \Omega_d(d, v \upharpoonright |d|, v_{|d|}), \hat{\Omega}_q f d_0 \rangle_{\delta_v^v}$$

based on Equation 10.30.

Decision wait generating function

From here it is a short step to define MDW : $\mathbb{N}^* \rightarrow \mathbb{H}$ as the multidimensional decision wait generating function

$$\text{MDW} = (\hat{\Omega}_d \hat{\Upsilon}_q) \circ \hat{\Upsilon}_d$$

analogous to Equation 10.29, potentially improving on Equation 10.21 for some strategically chosen $\hat{\Upsilon}_q$ and $\hat{\Upsilon}_d$. For a known metric of interest $\|\cdot\|$, the best choice of $\hat{\Upsilon}_d$ would be

$$\hat{\Upsilon}_d = \lambda f. \lambda s. (\lambda(m, t). t) \min\{(\|(\hat{\Omega}_d f) t\|, t) \in \mathbb{R} \times \mathbb{S} \mid (\psi \Delta_d) t = s\}$$

which can be optimized only with respect to a fixed choice of $f = \hat{\Upsilon}_q$ as noted previously.

10.6.4 Crossbar

Global crossbar decomposition strategies optimized with respect to a fixed quadrangular decomposition strategy are also a possibility, and may be an acceptable alternative to the method in Section 10.6.3 if using the crossbar in place of the dendriform decomposition is the only desired alteration. A global crossbar decomposition $u = ((p, d), v) \in \mathbb{S}$ meets the same condition $d_0 = (\psi \Delta_d^t) u$ captured by Equation 10.31 for its terminal nodes, and $|b d| = (\psi \Delta_c) u$ in general, where Δ_c is defined as

$$\Delta_c = \lambda r. (\Delta_d^t r) + (\Delta_c^n r) \quad (10.33)$$

for a function Δ_c^n given by

$$\Delta_c^n = \lambda(p, d, s). \langle 0, \delta_{|p_0|}^{|b d|}((\lambda k. \delta_k^s)((\lambda i. d_0 \upharpoonright \langle i \rangle)^* d_1) \upharpoonright (d_1 \prod d_0)) \rangle_{\delta_{|p|}^{|b d|+1} \delta_2^{|d|}} \quad (10.34)$$

constraining the dimensions the first $|d_1|$ subtrees $v \upharpoonright |d_1|$ to those required of the front end stages in a crossbar decision wait parameterized by d , and those of the remaining $\prod d_0 = |v| - |d_1|$ subtrees to d_1 , the dimensions required of the back end stages. These conditions lead directly to the concept of a **global crossbar decomposition strategy** defined as any function $\hat{\Upsilon}_c : \mathbb{N}^* \rightarrow \mathbb{S}$ satisfying

$$\forall s \in (\mathbb{N} - \{0\})^* - \{\epsilon\}. s = (\psi \Delta_c) \hat{\Upsilon}_c s$$

where the optimal choice is

$$\hat{\Upsilon}_c = \lambda f. \lambda s. (\lambda(m, t). t) \min\{(\|(\hat{\Omega}_c f) t\|, t) \in \mathbb{R} \times \mathbb{S} \mid (\psi \Delta_c) t = s\}$$

with respect to a fixed metric, a fixed quadrangular decomposition strategy $f = \hat{\Upsilon}_q$, and a combining form

$$\hat{\Omega}_c = \lambda f. \Lambda \lambda((p, d), v). (\phi p) \langle \Omega_c(d, v \upharpoonright |d_1|, v \ll |d_1|), \hat{\Omega}_q f d_0 \rangle_{\delta_v^v}.$$

The corresponding multidimensional decision wait generating function MDW : $\mathbb{N}^* \rightarrow \mathbb{H}$ is then

$$\text{MDW} = (\hat{\Omega}_c \hat{\Upsilon}_q) \circ \hat{\Upsilon}_c.$$

10.6.5 General

Nothing prevents a multidimensional decision wait from being built of planar building blocks whose decompositions vary depending on their contexts, so the restriction to multidimensional decomposition strategies parameterized by fixed planar decomposition strategies mentioned in [Section 10.6.3](#) and [Section 10.6.4](#) is not at all natural or obligatory. While removing it in this last section of the chapter, we might also take the opportunity to eliminate the need mentioned at the end of [Section 10.4.2](#) to choose exclusively between dendriform and crossbar decompositions.

A decomposition $u \in \mathbb{S}$ describing a multidimensional decision wait with dimensions $s \in \mathbb{N}^*$ whose building blocks are less constrained than in previous constructions need only satisfy

$$s = (\psi \Delta_g) u$$

by [Equation 10.26](#) for a function $\Delta_g : \mathbb{N}^{**} \times \mathbb{N}^{**} \times \mathbb{N}^{**} \rightarrow \{0, 1\}$ defined as

$$\Delta_g = \lambda r. (\Delta_d^n r) + (\Delta_c^n r) + (\Delta_q r) + (\Delta_g^t r) \quad (10.35)$$

based on [Equation 10.27](#), [Equation 10.32](#), [Equation 10.34](#), and the additional condition

$$\Delta_g^t = \lambda(p, d, s). (\lambda i. \langle 0, \delta_1^{|d_0|} \delta_{d_0}^{|p_1|} \rangle_i) \delta_1^{|d|} \delta_2^{|p|} \delta_s^\epsilon$$

pertaining to terminal trees describing 1-dimensional decision waits. This definition is reasonable because at most one of $\Delta_d^n r$, $\Delta_c^n r$, $\Delta_q r$ or $\Delta_g^t r$ can ever be non-zero for any $r = (p, d, s)$. It would be straightforward to build a decision wait given a decomposition $u \in \mathbb{S}$ having this property according to a recurrence $\hat{\Omega}_g = \Lambda \Omega_g : \mathbb{S} \rightarrow \mathbb{H}$ with

$$\Omega_g = \lambda((p, d), v). (\phi p) \begin{cases} \langle 1^{d_0}, \Omega_p(\sum d_0, \sum d_1) \rangle_{|d|-1} & \text{if } |v| = 0 \\ \Omega_d(d, v \uparrow |d|, v \ll |d|) & \text{if } |v| = |d| + 1 \\ \Omega_c(d, v \uparrow |d_1|, v \ll |d_1|) & \text{if } |v| = |d_1| + \prod d_0 \\ \Omega_q(d, v \uparrow 2, v \ll 2) & \text{if } |v| = (\sum d_0)(\sum d_1) + 2 \end{cases} \quad (10.36)$$

determining a decision wait $\hat{\Omega}_g u \in \mathbb{H}$ based on [Equation 10.18](#), [Equation 10.20](#), [Equation 10.25](#) and [Equation 10.22](#), because the conditions on $|v|$ in [Equation 10.36](#) are mutually exclusive for arguments $u \in \mathbb{S}$ with non-empty values of $(\psi \Delta_g) u$. We therefore define a general class of decision wait decomposition strategies as those functions $\hat{\mathcal{U}}_g : \mathbb{N}^* \rightarrow \mathbb{S}$ satisfying

$$\forall s \in (\mathbb{N} - \{0\})^* - \{\epsilon\}. s = (\psi \Delta_g) \hat{\mathcal{U}}_g s$$

whose optimal representative with respect to a fixed real valued metric is

$$\hat{\mathcal{U}}_g = \lambda s. (\lambda(m, t). t) \min\{(\|\hat{\Omega}_g t\|, t) \in \mathbb{R} \times \mathbb{S} \mid (\psi \Delta_g) t = s\}$$

and gives rise to the best multidimensional decision wait generating function of all.

$$\text{MDW} = \hat{\Omega}_g \circ \hat{\mathcal{U}}_g$$

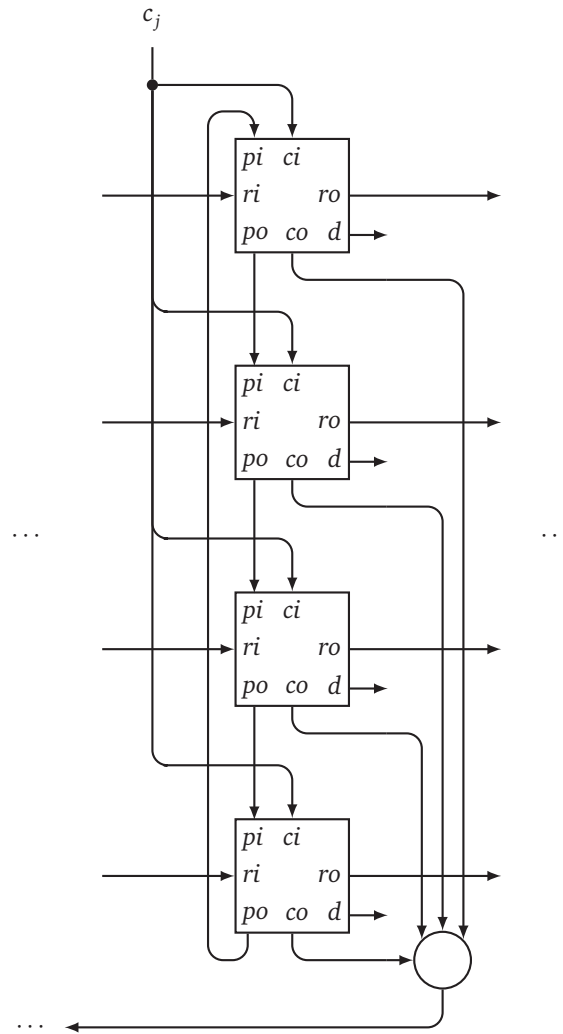
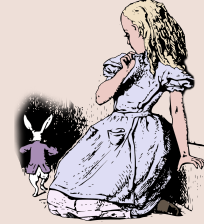


Figure 10.19: Broadcasting the column input concurrently to every cell in a cascading decision wait column might shave something off the critical path during the $ci-co$ handshake phase, but the rest still takes linear time (cf. Figure 10.6).

Decision support

1. Is there a higher dimensional analog to [Figure 10.2](#)?
2. What is the output permutation in [Figure 10.18](#)? Does it match [Equation 10.24](#)?
3. Solve $\hat{\Omega}_g(t) = I$ for t by [Equation 10.36](#).
4. Write a program that calculates the number of decompositions $u \in \mathbb{S}$ satisfying $s = (\psi \Delta_g) u$ for a given input list of dimensions $s \in \mathbb{N}^*$. For how large of a decision wait is an unguided exhaustive search for an optimum decomposition feasible? (hint: Generalize [Equation 12.18](#).)
5. [Figure 10.19](#) leads to an unexplored corner of the design space, cheaper than a quadrangular decision wait but maybe faster than a cascading one, where some or all of the cells in a column can be grouped together to receive their ci inputs concurrently and to synchronize their co outputs.
 - a) To cut costs, redesign this column to have two groups of two consecutive cells each such that both sets receive their ci concurrently but co propagates sequentially within them. What is the reduction in cost?
 - b) How many other ways are there to separate the column into groups of consecutive cells? Which is the fastest and which is the cheapest? How many would there be in a column of n cells, and how many in total for m independently decomposed columns of n cells each?
 - c) Make up a convention for identifying a given decomposition $d \in \mathbb{N}^{**}$ with a particular way of grouping all of the cells in an n -by- m cascading decision wait.
 - d) Make up an example of a decomposition strategy $\mathcal{U}_p : \mathbb{N}^2 \rightarrow \mathbb{N}^{**}$ that takes dimensions $\langle n, m \rangle \in \mathbb{N}^2$ to a decomposition $d = \mathcal{U}_p \langle n, m \rangle \in \mathbb{N}^{**}$ specifying a cascading decision wait following the convention in part c). Is there any way and any reason to make \mathcal{U}_p prioritize the speed of the higher numbered columns?
 - e) Upgrade the cascading decision wait combining form defined in [Equation 10.11](#) to a new version $\Omega_p : \mathbb{N}^{**} \rightarrow \mathbb{H}$ whereby a decision wait generating function could be defined as $Dw(n, m) = \Omega_p \mathcal{U}_p \langle n, m \rangle$ to implement the decomposition as indicated in part d). (N.B. This is the hard part.)
 - f) Define a function Δ_p analogous to Δ_q ([Equation 10.27](#)) but for global cascading decompositions, and upgrade [Equation 10.35](#) and [Equation 10.36](#) accordingly to enable optimization over a broader class of decompositions.



Never increase, beyond what is necessary, the number of entities required to explain anything.

William of Ockham

CHAPTER

11

THIN ON THE GROUND

The cost of a decision wait sometimes can be drastically reduced by permanently eliminating the output terminals and all relevant circuitry associated with any unwelcome input combinations. For example, with only three valid combinations of inputs instead of four, an LJOIN does its job using less than half the number of components of a 2-by-2 decision wait (cf. [Figure 9.13](#), [Figure 10.3](#) and [Figure 10.4](#)). In an environment where only three combinations were needed, it would be wasteful to do otherwise. Savings like these generalize to larger dimensions when the specifications permit, and can make sparse decision waits viable in many applications where an ordinary (*i.e.*, dense) decision wait would be suboptimal or prohibitive.

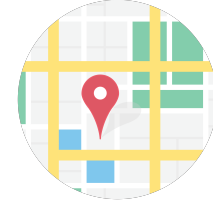
Designing a sparse decision wait of arbitrary dimensions using no more hardware than necessary has nothing but upside, but is infeasibly difficult to do manually. Fortunately this task can be automated based on techniques explored in this chapter, taking yet another load off the designer's mind and fitting into an overall theme of progressing toward higher level circuit synthesis, especially insofar as sparse decision waits have applications to state based circuit synthesis ([Chapter 15](#)) and to certain transcoding problems ([Chapter 13](#)). Moreover, readers who consider [Chapter 10](#) too easy should find this chapter more agreeable, which develops sparse decision waits in a way that reduces to dense decision waits as a special case. However, the presentation is ordered similarly where possible to keep the simpler material self contained towards the beginning for the sake of whoever is not in for the whole shooting match.

One reason to regard dense decision waits as special cases of sparse decision waits is that the interface between a dense decision wait and its environment is always fully described by a list of dimensions $s \in \mathbb{N}^*$ implying a total of $\sum s$ inputs organized into $|s|$ input buses, and $\prod s$ outputs. It may also be recalled from [Chapter 10](#) that any ensemble of $|b|$ input signals whose i -th is transmitted



along the b_i -th line of the i -th input bus for $b \in \mathbb{N}^{|s|}$ causes an output from the $((\bar{t}_s)^{-1} b)$ -th output terminal, provided that b_i is less than s_i for all $0 \leq i < |s|$.

The same can not be said of sparse decision waits with dimensions s in general. The effect of an ensemble of inputs $b = \langle 1, 1 \rangle$ to an LJOIN, for example, is undefined because there is no output in row 1 and column 1 as shown in Figure 9.13, even though it is reasonable to associate the dimensions of $s = \langle 2, 2 \rangle$ with an LJOIN and the inputs b are within this range. To express conditions like this one, we need to adopt a convention of specifying a sparse decision wait by its **coordinates** rather than by its dimensions alone. If an LJOIN is defined as the sparse decision wait whose coordinates are $c = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle\} \in \mathcal{P}(\mathbb{N}^{|s|})$, then there is no uncertainty about which combinations of inputs are allowed. We may loosely refer to a set $c \in \mathcal{P}(\mathbb{N}^*)$ hereafter as the coordinates of a sparse decision wait, or refer more precisely to each member $b \in \mathbb{N}^*$ of c as a **point** and each term $b_i \in \mathbb{N}$ of b as a coordinate when such distinctions are useful. The output arity of a sparse decision wait is not necessarily the product $\prod s$ of its dimensions s , but is given directly by the number $|c|$ of points, which is generally less. A dense decision wait with coordinates c and dimensions s satisfies $c = \mathcal{R}(\bar{t}_s)$ by Equation 10.3.



The rest of this chapter starts with a discussion of some unavoidable mathematical notation and conventions in Section 11.1, to be followed by a general treatment of sparse decision wait permutations and rotational transformations in Section 11.2, these being a prerequisite to some of the basic combining forms. For readers in a hurry, Section 11.3 describes a general sparse decision wait generating function as a simple recurrence that works for any specification but is usually suboptimal. Following in Section 11.4 are improvements to this basic construction that should yield somewhat better results for the special case of planar sparse decision waits if a bit more math is tolerable. Section 11.5 provides an alternative for multidimensional sparse decision waits by generalizing the dendriform and crossbar decompositions from Section 10.4 to them, and Section 11.6 adapts the idea of decomposition strategies introduced in Chapter 10 to sparse decision waits with a view to global optimality. Section 11.7 concludes with some remarks on how far these ideas can be trusted.

11.1 Notation

Specifying sparse decision waits by sets of points as proposed above requires all sorts of recurrences, partitions, transformations and miscellaneous manipulations of coordinates to generate circuits from them. Several notational devices introduced in this section that appear constantly in the sequel pertain to a form of ordinal functions, various concepts of a transpose, and a flattening operation.

11.1.1 Ordinals

The notation S° introduced in Section 5.1.4 for the ordinal function on a set S is used frequently in this chapter and subsequently. The same conventions noted previously with regard totally ordered sets continue to apply, but are extended henceforth to entail the following conditions pertaining to sets of lists S^* where S is totally ordered.

- No list precedes itself.
- The empty list ϵ precedes any non-empty list.

- A list $u \in S^*$ precedes a list $v \in S^*$ if either u_0 precedes v_0 , or u_0 is equal to v_0 and $u \ll 1$ precedes $v \ll 1$.

In other words, the implicit total ordering on lists of a totally ordered set is lexicographic.

A further idiomatic twist is to use this notation in an expression of the form $S^{\circ-1}$, which is the inverse function of S° in that it takes any number ranging from 0 through $|S| - 1$ to a unique member of S . Because lists are modeled as functions, this expression can also be interpreted as the list of length $|S|$ containing all elements of S in ascending order, and is treated as such hereafter.

11.1.2 Transposing

The expression x^\top , read “ x transpose”, denotes the transpose of an entity x when x is any expression devolving either to a set of lists, a list of lists, or a list of sets. The interpretation varies depending on the type of x .

Sets of lists

For a set of lists $c = \{\langle m, n, o \rangle, \langle p, q, r \rangle, \langle s, t, u \rangle\}$, the transpose $c^\top = \langle \{m, p, s\}, \{n, q, t\}, \{o, r, u\} \rangle$ is the list whose i -th term is the set of all i -th terms of members of c . This transformation is useful only when every member of c has the same length, so c^\top is identified with the empty list otherwise by definition. Specifically, for a set of lists $c \in \mathcal{P}(S^*)$, the list of sets $c^\top \in \mathcal{P}(S)^*$ is given by

$$c^\top = (\lambda n. (\lambda i. (\mu \lambda b. b_i) c)^* \iota_n) (\lambda s. \langle 0, \max s \rangle_{\delta_1^{|s|}}) (\mu \lambda b. |b|) c \quad (11.1)$$

using the μ operator defined by [Equation 5.1](#), and the maximum with respect to the usual ordering on natural numbers. Whenever c is a set of lists of equal length, which should be clear from the context, we can write c_i^\top for $(c^\top)_i$ without ambiguity. If c contains the coordinates for a sparse decision wait, then $|c^\top|$ is its number of dimensions, and $|c_i^\top|$ is the length of the i -th dimension.

Lists of lists

The transpose of a list $d \in S^{**}$ of equal-length lists to a list $d^\top \in S^{**}$ is completely analogous.

$$d^\top = (\lambda n. (\lambda j. (\lambda i. s_{ij})^* \iota_{|s|})^* \iota_n) (\lambda s. \langle 0, \max s \rangle_{\delta_1^{|s|}}) \mathcal{R}((\lambda b. |b|)^* d)$$

For example, $\langle \langle m, n, o \rangle, \langle p, q, r \rangle, \langle s, t, u \rangle \rangle^\top$ is $\langle \langle m, p, s \rangle, \langle n, q, t \rangle, \langle o, r, u \rangle \rangle$. This operation is its own inverse.

Lists of sets

It might seem appropriate to define the transpose of a list of sets as the inverse of the transpose of a set of lists, but that operation lacks a unique inverse. A more practical alternative for a list of sets $l \in \mathcal{P}(S)^*$ is a set $l^\top \in \mathcal{P}(S^*)$ containing every possible list of length $|l|$ whose n -th term is a member of the n -th term of l .

$$l^\top = (\mathcal{F}_{\{\epsilon\}} \lambda(h, t). (\mu \lambda(i, j). i : j) (h \times t)) l$$

This operation would satisfy $c \subseteq c^{\top\top}$ for a set $c \in \mathcal{P}(S^*)$ of lists of equal length. Another example is

$$\langle \mathcal{R}(t_a), \mathcal{R}(t_b), \mathcal{R}(t_c) \rangle^\top = \mathcal{R}(\bar{t}_{\langle a, b, c \rangle})$$

for natural numbers $a, b, c \in \mathbb{N}$ (cf. [Equation 10.3](#)).

11.1.3 Flattening

The list flattening operator $\flat : S^{**} \rightarrow S^*$ defined on lists of lists by Equation 8.11 suggests a comparable operation on lists of sets. If the members of a set $s \in \mathcal{P}(S)$ are totally ordered, then s induces a list $s^{\circ-1} \in S^*$ as proposed in Section 11.1.1. Transforming each set $x_i \subseteq S$ in a list of sets $x \in \mathcal{P}(S)^*$ to the list $x_i^{\circ-1} \in S^*$ results in a list of lists $(\lambda s. s^{\circ-1})^* x \in S^{**}$, which can then be flattened by \flat . An operator denoted

$$\overset{\circ}{\flat} : \mathcal{P}(S)^* \rightarrow S^*$$

and defined as follows encapsulates this transformation more briefly hereafter.

$$\overset{\circ}{\flat} = \flat \circ (\lambda s. s^{\circ-1})^* \quad (11.2)$$

For example, $\mathcal{R}(\overset{\circ}{\flat} x) = \bigcup \mathcal{R}(x)$ expresses the union of a list x of sets, and $|\overset{\circ}{\flat} x|$ is equal to $|\mathcal{R}(\overset{\circ}{\flat} x)|$ if all sets x_i in the list are mutually disjoint.

11.1.4 Coordinates

Two operations specific to sets of sparse decision wait coordinates are ubiquitous enough to warrant particular notations. These notations refer to the dimensions of a sparse decision wait with known coordinates, and to the local renumbering of a list of sparse decision wait specifications.

Dimensions

The coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ of a sparse decision wait determine a list of dimensions $\sigma c \in \mathbb{N}^*$ under one condition. An empty row in a sparse decision wait (*i.e.*, with no outputs in it), would mean no signal is ever allowed on the input terminal for that row, and therefore that terminal plays no role in the operation of the device. Because this case can be neglected without loss of generality, the set $a = c_i^{\top} \in \mathcal{P}(\mathbb{N})$ of input terminal numbers along the i -th dimensional axis can be assumed to contain every number from zero through $|a| - 1$ and be non-empty. A definition of $\sigma : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathbb{N}^*$ contingent on valid sets of coordinates is given by

$$\sigma = \lambda c. (\lambda d. \langle \epsilon, d \rangle_{\delta_{\epsilon}^{\dagger\{0\}}}) (\lambda a. |a| \delta_{|a|}^{\uparrow a})^* c^{\top} \quad (11.3)$$

in that σc is empty if the condition fails.

Local renumbering

The other operation is frequently applicable to subsets of the coordinates of a sparse decision wait that feature in its decomposition. If the rows and columns of a sparse decision wait are partitioned into multiple regions each representing a sparse decision wait to be generated separately and then glued together somehow, then some of the regions might exhibit non-consecutive sets of row or column coordinates contrary to the condition above. For sparse decision wait coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ and any list $d \in \mathcal{P}(\mathbb{N}^*)^*$ of regions satisfying $\bigcup \mathcal{R}(d) = c$, this issue would be resolved easily by renumbering points $b \in s$ locally relative to each region $s \in \mathcal{R}(d)$. That is, for each dimension $i < |b|$, a renumbered coordinate b_i to something in the range of 0 through $|s_i^{\top}| - 1$ without affecting their order is expressible as $(s_i^{\top})^{\circ} b_i$, its ordinal with respect to the set of all coordinates b_i in members b of the region s . The renumbered list ηd follows from this definition of $\eta : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^*)^*$.

$$\eta = (\lambda s. (\mu \lambda b. (\lambda i. (s_i^{\top})^{\circ} b_i)^* \iota_{|b|}) s)^* \quad (11.4)$$

11.2 Sparse decision wait transformations

Aside from their potential benefits for optimization, permutations and rotational transformations of sparse decision waits analogous to those of dense decision waits described in [Section 10.5](#) are needed in various combining forms. For sparse decision waits, the constructions are more complicated because they depend not just on the dimensions but specifically on the coordinates, and so are expressed as a function

$$\varphi(p, c) : \mathbb{H} \rightarrow \mathbb{H}$$

parameterized by a list of permutations $p \in \mathbb{N}^{**}$ and a set $c \in \mathcal{P}(\mathbb{N}^*)$ in terms of a composition

$$\varphi(p, c) = (\lambda(u, q). ((\varphi_1 u) (\hat{\varphi}_0 q) c) \circ (\varphi_0 q) c) (p_0, p \ll 1) \quad (11.5)$$

where φ_1 effects the rotation of the axes, φ_0 effects the permutations along each axis, and $\hat{\varphi}_0$ models the effect on the coordinates c due to the permutations q on the axes, which have an indirect effect on the result of φ_1 . This result is well defined only if there is one permutation in the list p for the rotation and one for each dimension

$$|p| = 1 + |\sigma c|$$

with the rotational permutation having a length matching the number of dimensions first in the list

$$|u| = |p_0| = |\sigma c|$$

and the rest having a length matching the j -th dimension for each $0 \leq j < |\sigma c|$.

$$|q_j| = |p_{j+1}| = (\sigma c)_j$$

With these conventions established, it remains only to specify the functions $\hat{\varphi}_0$, φ_0 , and φ_1 appearing in [Equation 11.5](#) to complete the construction. These derivations occupy [Section 11.2.1](#), [Section 11.2.2](#) and [Section 11.2.3](#) respectively.

11.2.1 Coordinate transformations

When we rotate a k -by- l -by- m decision wait, we can expect a result with dimensions k , l , and m in some other order, even if the inputs along each axis are also permuted, but if a sparse decision wait x described by coordinates c is permuted or rotated to $\varphi(p, c) x$, then the coordinates of the result might differ in more complicated ways from those of x . In particular, for each coordinate b_j in a point $b \in c$, a point in the intermediate result $((\varphi_0 q) c) x$ after permutation but before rotation has a j -th coordinate $((q_j)^{-1})_{b_j}$, where $q = p \ll 1$ is the list of permutations relevant to this phase. The complete set of them would be

$$(\mu \lambda b. (\lambda j. ((q_j)^{-1})_{b_j})^* \iota_{|b|}) c \in \mathcal{P}(\mathbb{N}^*)$$

denoted as $(\hat{\varphi}_0 q) c$ in terms of the desired function

$$\hat{\varphi}_0 : \mathbb{N}^{**} \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*))$$

taking the liberty of specifying an empty result whenever c has dimensions incompatible with q .

$$\hat{\varphi}_0 = \lambda q. \lambda c. \langle \emptyset, (\lambda i. \langle \emptyset, (\mu \lambda b. (\lambda j. ((q_j)^{-1})_{b_j})^* \iota_{|b|}) c \rangle_i) \prod (\lambda k. \delta_{|qk|}^{(\sigma c)_k})^* \iota_{|q|} \rangle_{\delta_{|q|}^{|\sigma c|}} \quad (11.6)$$



Although it is not needed until [Section 11.6](#), it is worth taking the opportunity at this point to note the overall effect on the coordinates due to both transformations φ_0 and φ_1 . The difference between the intermediate result $v = (\hat{\varphi}_0 q) c$ and the coordinates of the final result $\varphi(p, c) x$ is that each point $b \in v$ must be reordered to $b \circ u$ where $u = p_0$ is the permutation specifying the rotation of the axes in [Equation 11.5](#). The complete set of them would be

$$(\mu \lambda b. b \circ u) v \in \mathcal{P}(\mathbb{N}^*)$$

when v reflects the effects of the other permutations q . For future reference let these coordinates be denoted $(\hat{\varphi} p) c$ in terms of a function $\hat{\varphi} : \mathbb{N}^{**} \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*))$ given by

$$\hat{\varphi} = \lambda p. (\hat{\varphi}_1 p_0) \circ \hat{\varphi}_0(p \ll 1) \quad (11.7)$$

and the effect of the rotation

$$\hat{\varphi}_1 = \lambda u. \lambda v. \langle \emptyset, (\mu \lambda b. b \circ u) v \rangle_{\delta_{|r|}^{|\sigma|v|}} \quad (11.8)$$

also defined to yield an empty result in cases of incompatible dimensions as a matter of technical convenience.

11.2.2 Permuting along the axes

The function $\varphi_0 : \mathbb{N}^{**} \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow (\mathbb{H} \rightarrow \mathbb{H}))$ for transforming a sparse decision wait $x \in \mathbb{H}$ with coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ to a sparse decision wait $((\varphi_0 q) c) x \in \mathbb{H}$ with coordinates $(\hat{\varphi}_0 q) c$ has the effect of permuting the inputs along the n -th dimensional axis according to the permutation $a = q_n$. The result $((\varphi_0 q) c) x$ can be pictured as x with a wrapper around it consisting of an input and an output permutation network. The input permutation network

$$(\varepsilon R) (\lambda a. a \times |a|)^* q$$

depends only on q and would be the same even for a dense decision wait as discussed in [Section 10.5](#).

The output permutation network is best to understand as a recurrence intuitively similar to the one depicted in [Figure 10.18](#) but more complicated in the details. A sparse decision wait could have a different number of outputs in each row and therefore output buses of different widths from different rows. If the columns are permuted by a permutation $r \in \mathcal{R}(q)$, and the i -th row has outputs only in columns $k \in \mathcal{P}(\mathbb{N})$ with $|k| < |r|$, then the output bus from the i -th row needs an output permutation network described by a permutation consistent in some sense with r but not identical to r . The projection $s_i = r \uparrow k$ containing only the terms of r corresponding to extant outputs in k would almost work, but it is not generally a permutation unless it is renumbered to

$$(\lambda t. \mathcal{R}(s_i)^\circ t)^* s_i$$

thereby maintaining the order of those that remain. Furthermore, to derive the permutation network for the whole decision wait, we have to consider not just the permutation of the i -th row, but those of all rows $0 \leq i < |s|$. Flattening a list of them into a single permutation $\dot{o} s \in \mathbb{N}^*$ entails offsetting each term t of the i -th permutation by the cumulative sum of the lengths $|b(s \uparrow i)|$ of its predecessors in the list as follows.

$$\dot{o} = \lambda s. b (\lambda i. (\lambda t. |b(s \uparrow i)| + \mathcal{R}(s_i)^\circ t)^* s_i)^* \iota_{|s|} \quad (11.9)$$

The treatment of sparse decision waits with arbitrarily many dimensions $n = |\sigma c|$ calls for a more general definition of the output permutation network in terms of a recurrence $o(q, c)$ as indicated above. A set $c \in \mathcal{P}(\mathbb{N}^n)$ with $n > 1$ for an n -dimensional sparse decision wait determines a list of $|c_0^\top|$ sets

$$d = (\mu \lambda b. b \ll 1)^* ((\pi \lambda b. b_0) c)^{\circ-1} \in \mathcal{P}(\mathbb{N}^{n-1})^* \quad (11.10)$$

each describing an $(n - 1)$ -dimensional sparse decision wait (subject to renumbering). For example, for a three dimensional sparse decision wait ($n = 3$), this expression would represent a list of planar sparse decision waits ordered by their plane indices. (See Equation 6.6 for a reminder about the partitioning operator π .) For the inductive case of the recurrence $o(q, c)$, we can assume a known solution $o(q \ll 1, e)$ for any $(|q| - 1)$ -dimensional $e \in \mathcal{R}(d)$ by Equation 11.10. Each of these solutions describes the output permutation network for one of these lower dimensional building blocks, and by similar reasoning to that of Section 10.5, the buses themselves should be permuted according to the higher dimensional permutation q_0 and then combined by \dot{o} per Equation 11.9 as proposed above.

$$\dot{o}(((\lambda e. o(q \ll 1, e))^* (\mu \lambda b. b \ll 1)^* ((\pi \lambda b. b_0) c)^{\circ-1}) \circ q_0)$$

A recurrence with this inductive case implies a base involving a set $c \in \mathcal{P}(\mathbb{N}^1)$ of one-dimensional points, with c_0^\top not necessarily forming a set of consecutive indices unless the whole decision wait is only one-dimensional (*i.e.*, a bus). More generally, c in the base case might represent the column indices of the outputs in just one row of some larger sparse decision wait. The relevant segment of the output permutation would then follow from the projection $q_0 \uparrow c_0^\top$ according to the discussion above, suggesting this definition for the recurrence overall.

$$o(q, c) = \begin{cases} q_0 \uparrow c_0^\top & \text{if } |c^\top| = 1 \\ \dot{o}(((\lambda e. o(q \ll 1, e))^* (\mu \lambda b. b \ll 1)^* ((\pi \lambda b. b_0) c)^{\circ-1}) \circ q_0) & \text{otherwise} \end{cases}$$

Putting both permutation networks together into a wrapper around an operand $x \in \mathbb{H}$ leads to a definition

$$\varphi_0 = \lambda q. \lambda c. \lambda x. \mathbf{C}_{|bq|} \langle (\mathcal{F} \mathbf{R}) (\lambda a. a \times |^{|a|})^* q, x \rangle \times o(q, c)$$

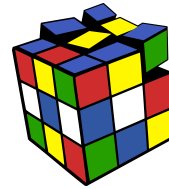
for this phase of the transformation φ in Equation 11.5.

11.2.3 Permuting the axes

The other phase $\varphi_1 : \mathbb{N}^* \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow (\mathbb{H} \rightarrow \mathbb{H}))$ that rotates a sparse decision wait $x \in \mathbb{H}$ with coordinates $v \in \mathcal{P}(\mathbb{N}^*)$ into a sparse decision wait $((\varphi_1 u) v) x$ with coordinates $(\hat{\varphi}_1 u) v$ also amounts to putting a wrapper of input and output permutation networks around the operand x .

The input permutation network specification is fairly straightforward with coordinates v implying dimensions σv and an ensemble of $|\sigma v|$ input buses with the j -th bus inside the wrapper having a width $(\sigma v)_j$. The given permutation u specifies a route for the i -th bus from the outside to the u_i -th position on the inside relative to the other buses, and hence an offset for its first line of $\sum((\sigma v) \uparrow s)$, the sum of the widths of the buses preceding it, where $s = u_i$ is the i -th term of u . The permutation describing the whole input permutation network is therefore

$$b (\lambda s. \iota_{(\sigma v)_s}^{\sum((\sigma v) \uparrow s)})^* u.$$



To affect a sparse decision wait with coordinates $(\hat{\varphi}_1 u) v$ by Equation 11.8, the output permutation network should run a wire to an externally visible terminal on $((\varphi_1 u) v) x$ pertaining to a point $b \in (\hat{\varphi}_1 u) v$, which appears at position $\mathcal{R}((\hat{\varphi}_1 u) v)^\circ b$ relative to the other externally visible output terminals. This wire should come from the internal output terminal on x due to the point $b \circ u^{-1}$, which appears at position $v^\circ(b \circ u^{-1})$ relative to the other output terminals on x . The whole output permutation is therefore the list of the internal output terminal positions on x ordered lexicographically by the coordinates b of their external destinations on $\varphi_1(u, v) x$.

$$(\lambda b. v^\circ(b \circ u^{-1}))^* ((\hat{\varphi}_1 u) v)^{\circ-1}$$

Putting the operand x together with its input and output permutations gives the following definition for φ_1

$$\varphi_1 = \lambda u. \lambda v. \lambda x. b (\lambda s. \iota_{(\sigma v)_s}^{\sum((\sigma v)^{1s})})^* u \times x \times (\lambda b. v^\circ(b \circ u^{-1}))^* ((\hat{\varphi}_1 u) v)^{\circ-1} \quad (11.11)$$

which completes the construction φ in Equation 11.5.

11.3 Fallback position

In this section we examine a couple of easy ways to construct sparse decision waits in case the rest of the chapter is too long or too much trouble. They may also be useful as a last resort when none of the more sophisticated techniques is appropriate. The degenerate sparse decision wait construction described in Section 11.3.1 relies in the worst case on building a dense decision wait and then ignoring some of the outputs, and the separable sparse decision wait discussed in Section 11.3.2 relies on dumb luck when the coordinate specification just happens to permit a decomposition into non-interacting parts.



The combination of these two constructions suffices for a basic sparse decision wait generating function $\text{SDW} : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathbb{H}$ satisfying the recurrence

$$\text{SDW}(c) = \begin{cases} (\lambda d. \Omega_s(d, \text{SDW}^* \eta d)) \mathcal{U}_s c & \text{if } \mathcal{U}_s c \neq \epsilon \\ \Omega_n(c, \text{MDW } \sigma c) & \text{otherwise} \end{cases} \quad (11.12)$$

in terms of any multidimensional decision wait generating function $\text{MDW} : \mathbb{N}^* \rightarrow \mathbb{H}$ defined in Chapter 10, where $\mathcal{U}_s : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)^*$ is the decomposition function taking sparse decision wait coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ to a list of separate coordinate specifications $\mathcal{U}_s c \in \mathcal{P}(\mathbb{N}^*)^*$.

- The separable combining form $\Omega_s : \mathcal{P}(\mathbb{N}^*)^* \times \mathbb{H}^* \rightarrow \mathbb{H}$ takes a list of separate specifications $d \in \mathcal{P}(\mathbb{N}^*)^*$ and a list of sparse decision waits $x \in \mathbb{N}^*$ where x_i implements d_i for all $0 \leq i < |d|$ to a sparse decision wait $\Omega_s(d, x) \in \mathbb{H}$ implementing the original specification c .
- The degenerate combining form $\Omega_n : \mathcal{P}(\mathbb{N}^*) \times \mathbb{H} \rightarrow \mathbb{H}$ simply takes the coordinates c and a dense decision wait with the same dimensions implied by c to a sparse decision wait with coordinates c .

coordinates c	permutations p
$\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle$	$\langle \iota_2, \langle 0, 1 \rangle, \langle 0, 1 \rangle \rangle$
$\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 1 \rangle$	$\langle \iota_2, \langle 0, 1 \rangle, \langle 1, 0 \rangle \rangle$
$\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle$	$\langle \iota_2, \langle 1, 0 \rangle, \langle 0, 1 \rangle \rangle$
$\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle$	$\langle \iota_2, \langle 1, 0 \rangle, \langle 1, 0 \rangle \rangle$

Table 11.1: permutation lists $p \in \mathbb{N}^{**}$ needed to synthesize a 2-by-2 sparse decision wait with coordinates $c \in \mathcal{P}(\{0, 1\}^2)$ as $\varphi(p, \{0, 1\}^2 - \{1\}^2)$ LJOIN by Equation 11.5 and Equation 9.18

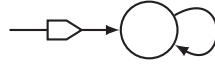


Figure 11.1: The useless circuit $\mathbf{Z}^2\mathbf{R}(\text{PUSH}, \text{JOIN})$ has one input terminal and no outputs but lays a trap for any signal coming its way during simulation.

11.3.1 Degenerate

If the only sparse decision wait ever required were an LJOIN, then the degenerate case combining form Ω_n would be especially easy because its result would be given already by Equation 9.18. Moreover, any 2-by-2 sparse decision wait with three outputs can always be implemented by some permutation of an LJOIN as shown in Table 11.1. That is, interchanging the rows, columns, or both as needed relocates the missing output in the lower right corner of Figure 9.13 to any desired corner. The coordinates

$$\{0, 1\}^2 - \{1\}^2 = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle\}$$

supplied in the operand to φ are required to match the actual coordinates of an LJOIN. Expressing p in terms of c consistently with Table 11.1 is a matter of code golf, but

$$p = \langle \iota_2, (c^{\circ-1})_1, (\lambda i. \langle \delta_2^i, \delta_1^i \rangle) \sum (c^{\circ-1})_1^T \rangle$$

is good enough for our purposes.

In all other cases, a sparse decision wait with coordinates c is expressible in terms of a dense decision wait with dimensions σc and the outputs associated with the unused input combinations suppressed, which are the input combinations $b \in \mathcal{R}(\bar{\iota}_{\sigma c}) - c$. Rather than just making a mental note of them, we can indicate explicitly the outputs that are suppressed by connecting an instance of the circuit shown in Figure 11.1 to each one of them as shown in Figure 11.2. This bookkeeping step is necessary to reduce the number of observable outputs on the circuit from $\prod \sigma c$ to the correct value of $|c|$ for a sparse decision wait with coordinates c . It also has the effect of prohibiting the input combinations leading to any of these outputs in the sense that the Petri net model of the circuit attains an unsafe marking according to the theory developed in Part II of this book. This condition is desirable because it ensures that events whose effects are unspecified never go unnoticed during formal verification.

To implement the degenerate sparse decision wait in general, we envision the dense decision wait $x \in \mathbb{H}$ with dimensions σc connected by a bus of width $\prod \sigma c$ to an array of $\prod \sigma c$ blocks in



Figure 11.2: If there were no better way to implement an LJOIN, it could be made from a dense decision wait of similar dimensions with the outputs due to invalid input combinations suppressed.

which the i -th block is a wire if $b = (\bar{t}_{\sigma c})_i$ is a member of c , but is $\mathbf{Z}^2\mathbf{R}(\text{PUSH}, \text{JOIN})$ otherwise.

$$\mathbf{C}_{\prod \sigma c} \langle x, (\mathcal{F}\mathbf{R}) (\lambda b. \langle \mathbf{Z}^2\mathbf{R}(\text{PUSH}, \text{JOIN}), \mathbf{I} \rangle_{\delta_c^{\{b\} \cup c}})^* \bar{t}_{\sigma c} \rangle$$

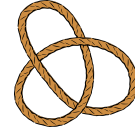
A definition for the degenerate sparse decision wait combining form Ω_n follows directly from the two alternatives considered above.

$$\Omega_n(c, x) = \begin{cases} \varphi(\langle \iota_2, (c^{\circ-1})_1, (\lambda i. \langle \delta_2^i, \delta_1^i \rangle) \sum (c^{\circ-1})_1^i, \{0, 1\}^2 - \{1\}^2 \rangle) \text{LJOIN} & \text{if } \delta_3^{|c|} \delta_{\emptyset}^{c - \{0, 1\}^2} = 1 \\ \mathbf{C}_{\prod \sigma c} \langle x, (\mathcal{F}\mathbf{R}) (\lambda b. \langle \mathbf{Z}^2\mathbf{R}(\text{PUSH}, \text{JOIN}), \mathbf{I} \rangle_{\delta_c^{\{b\} \cup c}})^* \bar{t}_{\sigma c} \rangle & \text{otherwise} \end{cases}$$

This method is of course quite an inept way of constructing almost any sparse decision wait other than an LJOIN without a few optimizations to come.

11.3.2 Separable

One such optimization that is always to be preferred where applicable is illustrated in Figure 11.3. If the outputs from a decision wait can be separated into regions having no rows or columns in common, then each region can be implemented separately with nothing but suitably chosen permutation networks needed to combine them (at no cost). This optimization generalizes to any number of dimensions, and does not require the regions to be contiguous.



Identifying the separable regions from a set of coordinates is equivalent to identifying the connected components in an undirected graph, a well understood problem solvable by efficient algorithms [65, 139]. Each hyperplane perpendicular to the major dimension corresponds to a node in the graph, and any two nodes are connected by an edge whenever there is an output at the same position in both of them. For example, in a planar sparse decision wait, each node is a row and two nodes are connected whenever both of them have an output in the i -th column for at least one i .

Decomposition function

To specify an algorithm for decomposing a specification given by a set of coordinates $c \in \mathcal{P}(\mathbb{N}^*)$, let a function

$$f = \lambda b. \{t \in \mathbb{N}^* \mid b_0 : t \in c\}$$

map any point $b \in c$ to the major hyperplane containing b expressed as a set of lower dimensional coordinates t , which we envision as a node in a graph as discussed above. Then the region of the graph including the hyperplane $f b$ and any hyperplanes directly adjacent to it is

$$\{m \in c \mid f b \cap f m \neq \emptyset\} = (\lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) b$$

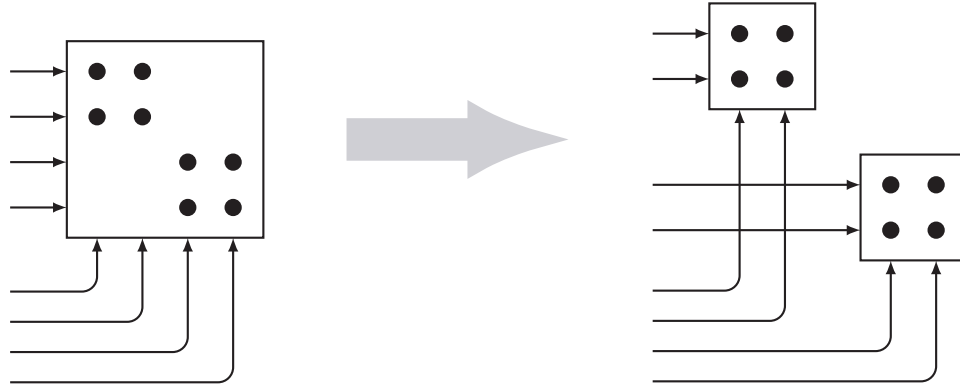


Figure 11.3: an example of a separable sparse decision wait

and the extension of this region to all hyperplanes connected directly or indirectly to the one containing b is the percolation

$$(\rho \lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) \{b\}$$

by Equation 6.4. A partition of the whole graph into similarly extended regions by Equation 6.6

$$(\pi \rho \lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^*))$$

determines a list thereof ordered by their major coordinate index

$$((\pi \rho \lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) c)^{\circ-1} \in \mathcal{P}(\mathbb{N}^*)^*$$

suggesting a definition for the separable decomposition function along the lines of

$$\lambda c. (\lambda f. ((\pi \rho \lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) c)^{\circ-1}) \lambda b. \{t \in \mathbb{N}^* \mid b_0 : t \in c\}$$

unless this formula yields a unit list, meaning the graph has only one connected component and the specification c therefore lacks a useful separable decomposition. As a matter of technical convenience the result in this case is the empty list by definition.

$$\mathcal{U}_s = (\lambda d. \langle d, \epsilon \rangle_{\delta_1^{|d|}}) \circ \lambda c. (\lambda f. ((\pi \rho \lambda n. \{m \in c \mid f n \cap f m \neq \emptyset\}) c)^{\circ-1}) \lambda b. \{t \in \mathbb{N}^* \mid b_0 : t \in c\}$$

Inverse decomposition function

Given a non-empty decomposition $d = \mathcal{U}_s c$, we can always recover the coordinates $c = \bigcup \mathcal{R}(d)$, as we need to do shortly when deriving the combining form Ω_s , but \mathcal{U}_s is only the first of several decompositions to be developed in this chapter, and when it comes to defining sparse decision wait generating functions by cases, it is more helpful to have an explicit inverse \mathcal{U}_s^{-1} for which

$$c = \mathcal{U}_s^{-1} d$$

does not hold unless $d = \mathcal{U}_s c$ holds. To meet this condition, the regions $e \in \mathcal{R}(d)$ each inducing a list of sets of input indices $e^\top \in \mathcal{P}(\mathbb{N})^{|\sigma c|}$ must collectively determine a list of lists of sets

$$a = ((\lambda e. e^\top) * d)^\top \in (\mathcal{P}(\mathbb{N})^{|\sigma c|})^{|\sigma c|} \quad (11.13)$$

wherein the i -th term $a_i \in \mathcal{P}(\mathbb{N})^{|\sigma c|}$ is a list of mutually disjoint sets covering c_i^\top for all $0 \leq i < |\sigma c|$, and d must have a length greater than 1. In other words,

$$(\lambda a. \langle 0, \delta_\epsilon^{t_2^\dagger \{ |d| \}} \prod (\lambda i. \delta_{c_i^\top}^{\bigcup \mathcal{R}(a_i)} \delta_{(\sigma c)_i}^{|\mathring{b} a_i|})^* \iota_{|a|} \rangle_{\delta_{|a|}^{|\sigma c|}}) ((\lambda e. e^\top) * d)^\top$$

would have to be non-zero, calling for a definition of \mathcal{U}_s^{-1} as follows.

$$\mathcal{U}_s^{-1} = \lambda d. (\lambda c. (\lambda j. \langle \emptyset, c \rangle_j) (\lambda a. \langle 0, \delta_\epsilon^{t_2^\dagger \{ |d| \}} \prod (\lambda i. \delta_{c_i^\top}^{\bigcup \mathcal{R}(a_i)} \delta_{(\sigma c)_i}^{|\mathring{b} a_i|})^* \iota_{|a|} \rangle_{\delta_{|a|}^{|\sigma c|}}) ((\lambda e. e^\top) * d)^\top) \bigcup \mathcal{R}(d)$$

Combining form

Being concerned with a list of separate non-interacting decision waits $x \in \mathbb{H}^*$ as shown in [Figure 11.3](#), a definition for the combining form Ω_s should feature only the parallel combination $(\mathcal{F} \mathbf{R}) x$ surrounded by an input permutation network and an output permutation network.

The output permutation is the easy one. The i -th externally visible output terminal is associated with a point $b = (c^{\circ-1})_i$, where $c = \mathcal{U}_s^{-1} d$ is the specification from which the separable decomposition $d = \mathcal{U}_s c$ is derived. The signal to this output terminal comes from the output terminal numbered

$$(\mathring{b} d)^{-1} b$$

on the array of building blocks $(\mathcal{F} \mathbf{R}) x$, because $\mathring{b} d$ expresses the list of points in order of their output terminal positions on the array. Hence the whole output permutation is simply

$$(\mathring{b} d)^{-1} * (\mathcal{U}_s^{-1} d)^{\circ-1}$$

in terms of d alone.

For the input permutation, we recall temporarily the list $a = ((\lambda e. e^\top) * d)^\top$ from [Equation 11.13](#). Each term a_i is a list of $|\sigma c|$ sets of input terminal numbers, with the j -th set pertaining to the j -th dimension of the i -th building block. The initial set a_{i0} therefore contains the actual line numbers of the lines to be connected to the i -th block along its major dimensional axis relative to the whole externally visible input bus. However, the numbers in a set a_{ij} with $j > 0$ must be interpreted as being relative to that part of the externally visible input bus connected only to inputs along the j -th dimensional axis. The absolute input bus line numbers associated with members of a_{ij} are obtained only by offsetting each of them with

$$\sum((\sigma c) \upharpoonright j)$$

which is the number of all preceding bus lines due to other dimensions, in an expression like

$$(\mu \lambda k. k + \sum((\sigma c) \upharpoonright j)) a_{ij}.$$

Stringing together the list of all input line numbers connected to the i -th block over all dimensions j

$$\mathring{b} (\lambda j. (\mu \lambda k. k + \sum((\sigma c) \upharpoonright j)) a_{ij})^* \iota_{|\sigma c|}$$

and doing the same for all building blocks associated with the decomposition d

$$\mathring{b} (\lambda i. \mathring{b} (\lambda j. (\mu \lambda k. k + \sum((\sigma c) \uparrow j)) a_{ij})^* \iota_{|\sigma c|})^* \iota_{|d|}$$

would result in the inverse of the required input permutation, because it lists the external line number for each internal terminal rather than the internal terminal number for each line per convention. The actual input permutation $p d \in \mathbb{N}^*$ is given by a function $p : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathbb{N}^*$ defined as

$$p = \lambda d. (\lambda c. (\mathring{b} (\lambda i. \mathring{b} (\lambda j. (\mu \lambda k. k + \sum((\sigma c) \uparrow j)) ((\lambda e. e^\top)^* d)_{ij}^\top)^* \iota_{|\sigma c|})^* \iota_{|d|})^{-1}) \mathcal{U}_s^{-1} d.$$

A definition for the combining form Ω_s follows easily from these two permutations

$$\Omega_s(d, x) = (\mathcal{F} \mathbf{R}) p d \times x \times (\mathring{b} d)^{-1*} (\mathcal{U}_s^{-1} d)^{\circ^{-1}}$$

and completes the specification of our first iteration of a sparse decision wait generating function SDW in Equation 11.12.

11.4 Planar sparse decision waits



A small investment of effort in the way of a few further planar decompositions yields a large improvement to the basic construction of Equation 11.12 in the way of efficiency. The vertical and horizontal *spanning* decompositions described in Section 11.4.1 separate the specification into two parts, and the *enmeshed* decomposition discussed in Section 11.4.2 employs three. The combination of these decompositions along with the separable decomposition leads to the more robust sparse decision wait generating function in Equation 11.14, which need never resort to the degenerate form for any planar sparse decision wait larger than 2-by-2.

$$\text{SDW}(c) = \begin{cases} (\lambda d. \Omega_s(d, \text{SDW}^* \eta d)) \mathcal{U}_s c & \text{if } \mathcal{U}_s c \neq \epsilon \\ (\lambda d. \Omega_v(d, \text{SDW}(\eta d)_0, \text{SDW}(\eta d)_1)) \mathcal{U}_v c & \text{if } \mathcal{U}_v c \neq \epsilon \\ (\lambda d. \Omega_h(d, \text{SDW}(\eta d)_0, \text{SDW}(\eta d)_1)) \mathcal{U}_h c & \text{if } \mathcal{U}_h c \neq \epsilon \\ (\lambda d. \Omega_e(d, \text{SDW} d_0, \text{SDW}^* \eta \langle d_1, d_2 \rangle, \text{SDW}^* \eta (d \ll 3)) \mathcal{U}_e c & \text{if } \mathcal{U}_e c \neq \epsilon \\ \Omega_n(c, \text{MDW } \sigma c) & \text{otherwise} \end{cases} \quad (11.14)$$

Even if the parts are not separable in the sense of Section 11.3.2, the other decompositions compensate through the use of routing circuitry that incurs no more than a logarithmic increase in path length.

11.4.1 Spanning

Figure 11.4 shows a high level schematic representation of a planar sparse decision wait obtained by vertical spanning combination. A horizontal spanning combination differs only in that the blocks x and y , which are planar decision waits, would appear side by side, with the roles of rows and columns interchanged. The intuition behind this construction is roughly as follows.

- For a planar sparse decision wait with coordinates $c \in \mathcal{P}(\mathbb{N}^2)$, a vertical spanning decomposition, if one exists, is determined by a proper subset $t \subset c$ covering a whole number of more than one but fewer than $|c_0^\top|$ rows such that the most populous row $r \subset t$ has an output in

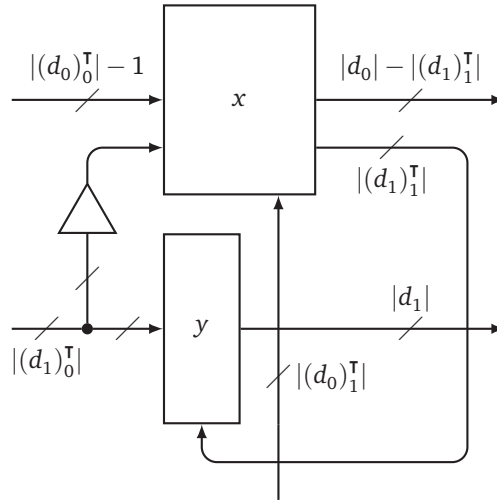


Figure 11.4: spanning combination $\Omega_v(d, x, y)$ where x and y have coordinates $(\eta d)_0$ and $(\eta d)_1$

every non-empty column of t . That is, the row r “spans” t , meaning this condition among others holds.

$$r_1^T = t_1^T$$

- The coordinates of y in Figure 11.4 are a locally renumbered version of t , and the coordinates of x are a locally renumbered version of $(c - t) \cup r$. That is, some chosen spanning row r is replicated in both x and y . Advantageous choices of t will probably satisfy $|t_1^T| < |c_1^T|$, corresponding to a block y with fewer columns than x .
- A completion detecting bus to the row inputs on y also triggers the input to row r on x , and a bus from row r on x drives the column inputs on y .
- Input and output permutation networks not depicted in the figure make the combination appear to have coordinates c from the outside.

Figure 11.5 shows a concrete example of a vertical spanning decomposition using these terms. The rest of this section elaborates on the details.

Decomposition functions

As the description above suggests, there is generally more than one way to obtain a spanning decomposition, and the same is true for all decompositions considered subsequently in this chapter. Rather than seeking a specific vertical spanning decomposition function \mathcal{U}_v , we define only a set valued function

$$\nabla_v : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^2)$$

taking a set $c \in \mathcal{P}(\mathbb{N}^*)$ to the set $\nabla_v c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^2)$ of all possible vertical spanning decompositions of a sparse decision wait with coordinates c . We may then write \mathcal{U}_v to denote any function of the

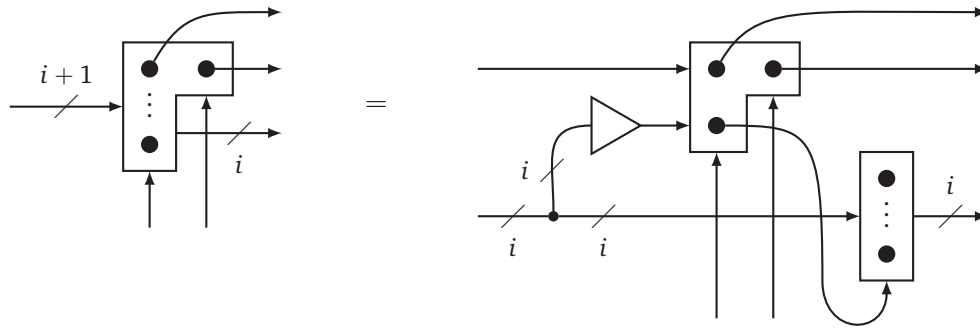


Figure 11.5: vertical spanning decomposition for coordinates $c = \mathcal{R}(\langle t_{i+1}, 0^{i+1} \rangle^T) \cup \{t_2\}$, a dense columnar subset $t = \mathcal{R}(\langle t_{i+1}^1, 0^i \rangle^T)$, and a spanning row $r = \langle 1, 0 \rangle$

reader's choice satisfying

$$\forall c \in \mathcal{P}(\mathbb{N}^2). \mathcal{U}_v c \in (\nabla_v c) \cup \{\epsilon\}. \tag{11.15}$$

This convention is especially relevant when the same coordinates c simultaneously permit horizontal, vertical, and enmeshed decompositions, because then the judicious choice of at most one of $\mathcal{U}_v c \neq \epsilon$, $\mathcal{U}_h c \neq \epsilon$, or $\mathcal{U}_e c \neq \epsilon$ to be true ensures a deterministic definition for SDW in Equation 11.14 respecting the reader's preferences.

It is straightforward to give a naive definition for ∇_v by starting with a function

$$p = \pi \lambda b. b_0$$

that partitions any set $c \in \mathcal{P}(\mathbb{N}^2)$ into a set of rows $p c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2))$. Then any proper subset of the rows $s \in \mathcal{P}(p c) - \{p c\}$ determines a set of points $t = \bigcup s$ covering a whole number of rows. If the set t covers at least two rows, then it stands a chance of having a proper subset $r \in (p t) - \{t\}$ satisfying $r_1^T = t_1^T$. If so, the pair (t, r) determines a member $\langle (c - t) \cup r, t \rangle$ of $\nabla_v c$, which therefore can be specified in full as follows.

$$\nabla_v = \lambda c. (\mu \lambda(t, r). \langle (c - t) \cup r, t \rangle) (\lambda p. \bigcup_{s \in (\mathcal{P}(p c) - \{p c\}) \cap \mathcal{P}(\mathbb{N}^2)} (\lambda t. (\{t\} \times \{r \in (p t) - \{t\} \mid r_1^T = t_1^T\})) \bigcup s) \pi \lambda b. b_0$$

The problem with this definition of ∇_v is that it might be difficult to compute efficiently if it depends on enumerating the power set of the rows. In the end at most one decomposition $\mathcal{U}_v c \in \nabla_v c$ can be selected for a particular c , so if an approximation of $\nabla_v c$ were to contain the decomposition of interest, then it might save time to compute only the approximation. A reasonable approximation follows from the assumptions that a choice of $t \subset c$ in a decomposition is always preferable to a proper subset $t' \subset t$ thereof, and that the lexicographically minimal spanning row $r \subset t$ is as good as any other. These assumptions admit only a single decomposition for the specification in Figure 11.5 for example, which would have the columnar decision wait as tall as possible, and generally would favor taller blocks y in Figure 11.4 over shorter ones with the same width.



One way of computing this approximation starts by making a list $v_0 c \in \mathcal{P}(\mathbb{N}^*)^*$ of the rows that are not fully populated in order of decreasing cardinality.

$$v_0 = \lambda c. (\mu \lambda(m, r). r)^* ((\mu \lambda r. (|c_1^\top| - |r|, r)) (\pi \lambda b. b_0) c) - (\{0\} \times \mathcal{P}(\mathbb{N}^*))^{\circ-1} \quad (11.16)$$

To find any additional rows e for which $e_1^\top \subseteq r_1^\top$ holds, only the rows after r in the list $v_0 c$ need to be searched, and having been found, they can be eliminated from further consideration. For a list of the form $r : z = v_0 c$, the set of all such rows is expressible as

$$\mathcal{R}(z \uparrow \mathcal{P}(\mathbb{N}^1 \parallel (r_1^\top)^1)) \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2))$$

this being the set of rows in the range of z containing only points in $\mathbb{N}^1 \parallel (r_1^\top)^1$, whose column indices therefore match a member of r_1^\top (cf. Equation 7.3 on concatenation of sets of lists). If there is at least one such row e , then the union t of all rows e with r determines a possible decomposition candidate. Denote this result $t = v_1(r : z)$ with $v_1 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^2)$ defined as

$$v_1 = \lambda(r : z). (\lambda t. \langle t, \emptyset \rangle_{\delta t}) (r \cup \mathcal{R}(z \uparrow \mathcal{P}(\mathbb{N}^1 \parallel (r_1^\top)^1))).$$

Any decomposition candidates not covered by t in the rest of the list z can be found more quickly with all subsets of t filtered out of it

$$z \uparrow (\mathcal{P}(\mathcal{P}(\mathbb{N}^2)) - \mathcal{P}(t))$$

suggesting a recursively defined function $v_2 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}^2))) \times \mathcal{P}(\mathcal{P}(\mathbb{N}^2))$

$$v_2 = \lambda w. \begin{cases} \emptyset & \text{if } w = \epsilon \\ (\lambda t. \{(t, w_0)\} \cup v_2((w \ll 1) \uparrow (\mathcal{P}(\mathcal{P}(\mathbb{N}^2)) - \mathcal{P}(t)))) v_1 w & \text{otherwise} \end{cases}$$

whereby all pairs $(t, r) \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2)) \times \mathcal{P}(\mathcal{P}(\mathbb{N}^2))$ of maximal sets of rows t with their respective spanning rows r follow as $v_2 v_0 c$. A set of vertical spanning decompositions

$$(\mu \lambda(t, r). \langle (c - t) \cup r, t \rangle) v_2 v_0 c$$

obtained by this line of reasoning however might not exclude the anomalous result $\langle c, \emptyset \rangle$, so the restriction to

$$v_3 (\mu \lambda(t, r). \langle (c - t) \cup r, t \rangle) v_2 v_0 c$$

for $v_3 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow (\mathcal{P}(\mathbb{N}^2) - \{\emptyset\})^*$ defined by

$$v_3 = \mu \lambda d. d \cap (\mathcal{P}(\mathbb{N}^2) - \{\emptyset\})^* \quad (11.17)$$

leads to a more suitable definition for the approximate alternative to ∇_v .

$$\nabla_v = \lambda c. v_3 (\mu \lambda(t, r). \langle (c - t) \cup r, t \rangle) v_2 v_0 c$$

In any case, a definition for the set of horizontal spanning decompositions $\nabla_h c$ follows immediately almost for free from a reversal of the coordinates before and after

$$\nabla_h = \lambda c. (\lambda f. (\mu f^*) \nabla_v f c) \mu f_\epsilon \lambda(h, z). z \parallel \langle h \rangle$$

with the decomposition function \mathcal{U}_h following a similar convention to that of Equation 11.15.

Inverse decomposition functions

Even though the choice of a decomposition function \mathcal{U}_v satisfying $\mathcal{U}_v c \in \nabla_v c \cup \{\epsilon\}$ is somewhat flexible, there are clear criteria for deciding whether a given list $d \in \mathcal{P}(\mathbb{N}^*)^*$ constitutes a valid vertical spanning decomposition for some coordinates $c \in \mathcal{P}(\mathbb{N}^2)$, so no need for more than one inverse decomposition function

$$\mathcal{U}_v^{-1} : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^2)$$

in the sense that it can satisfy

$$\forall c \in \mathcal{P}(\mathbb{N}^*), \forall d \in \nabla_v c. \mathcal{U}_v^{-1} d = c \quad (11.18)$$

regardless of how \mathcal{U}_v is chosen. These criteria include the number of building blocks $|d| = 2$, the number of dimensions $|\sigma c| = 2$ for coordinates $c = \bigcup \mathcal{R}(d)$, the intersection $d_0 \cap d_1$ occupying exactly one row $r \in (\pi \lambda b. b_0) c$ spanning d_1 , and non-separability implied by d_0 spanning c .

$$\mathcal{U}_v^{-1} = \lambda d. (\lambda r. c). \langle \emptyset, (\lambda p. (\lambda i. \langle \emptyset, c \rangle_i) \delta_{c_1^T}^{(d_0)_1^T} \delta_{r_1^T}^{(d_1)_1^T} \delta_p^{p \cup \{r\}}) (\pi \lambda b. b_0) c \rangle_{\delta_2^{|\sigma d|} \delta_2^{|\sigma c|}} (\bigcap \mathcal{R}(d), \bigcup \mathcal{R}(d))$$

The inverse horizontal spanning decomposition function is obtained analogously by a reversal of the coordinates.

$$\mathcal{U}_h^{-1} = (\lambda f. f \circ \mathcal{U}_v^{-1} \circ f^*) \mu f_e \lambda (h, z). z \parallel \langle h \rangle \quad (11.19)$$

Combining forms

The combining form $\Omega_v : \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \times \mathbb{H} \rightarrow \mathbb{H}$ appearing in Equation 11.14 is required to take a vertical spanning decomposition $d \in \mathcal{P}(\mathbb{N}^2)^2$, a block $x \in \mathbb{H}$, and a block $y \in \mathbb{H}$ to a result

$$\Omega_v(d, x, y) \in \mathbb{H}$$

with coordinates $c = \mathcal{U}_v^{-1} d$ as shown in Figure 11.5 provided c is non-empty, x has coordinates $(\eta d)_0$, and y has coordinates $(\eta d)_1$. A prominent feature of this construction is the completion detecting bus

$$\mathbf{F}_v \langle \text{FORK}^v \uparrow_2^1, \text{MERGE } v \rangle$$

where $v = |(d_1)_0^T|$ is the number of rows of y . A block $S_0(d, y) \in \mathbb{H}$ combining the completion detecting bus with y is expressible in terms of

$$S_0 : \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \rightarrow \mathbb{H}$$

defined as

$$S_0 = \lambda (d, y). (\lambda v. \mathbf{F}_v \langle \mathbf{F}_v \langle \text{FORK}^v \uparrow_2^1, \text{MERGE } v \rangle, y \rangle \uparrow 1) |(d_1)_0^T|$$

with the completion detection signal moved to the last output position. To connect x with this block, it would be helpful to have the spanning row input in r_0^T as the first input to x

$$((d_0)_0^T)^{\circ*} \overset{\circ}{b} \langle r_0^T, (d_0)_0^T - r_0^T \rangle \times x$$

where $r = d_0 \cap d_1$ is the set of spanning row coordinates, and the output bus from this row in the last position

$$((d_0)_0^T)^{\circ*} \overset{\circ}{b} \langle r_0^T, (d_0)_0^T - r_0^T \rangle \times x \times d_0^{-1*} \overset{\circ}{b} \langle d_0 - r, r \rangle$$

so that the last $|r|$ outputs from x reach the column inputs on y and the completion detector output can be rolled into alignment with the spanning row input on x in an expression like

$$\mathbf{L}_{|r|} \langle ((d_0)_0^\top)^{\circ*} \overset{\circ}{b} \langle r_0^\top, (d_0)_0^\top - r_0^\top \rangle \times x \times d_0^{-1*} \overset{\circ}{b} \langle d_0 - r, r \rangle, S_0(d, y) \rangle \Downarrow 1$$

to provide for a combined block with one further connection

$$\mathbf{Z}(\mathbf{L}_{|r|} \langle ((d_0)_0^\top)^{\circ*} \overset{\circ}{b} \langle r_0^\top, (d_0)_0^\top - r_0^\top \rangle \times x \times d_0^{-1*} \overset{\circ}{b} \langle d_0 - r, r \rangle, S_0(d, y) \rangle \Downarrow 1)$$

expressible more succinctly as $S_1(d, x) S_0(d, y)$ in terms of $S_1 : \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ given by

$$S_1 = \lambda(d, x). \lambda s. (\lambda r. \mathbf{Z}(\mathbf{L}_{|r|} \langle ((d_0)_0^\top)^{\circ*} \overset{\circ}{b} \langle r_0^\top, (d_0)_0^\top - r_0^\top \rangle \times x \times d_0^{-1*} \overset{\circ}{b} \langle d_0 - r, r \rangle, s \rangle \Downarrow 1) (d_0 \cap d_1).$$

It remains only to effect the required input and output permutations. The first externally visible inputs to the block derived up to this point reach the $|(d_0)_0^\top| - 1$ row inputs left exposed on x , the next $j = |(d_0)_1^\top|$ reach the column inputs on x , and the last $|(d_1)_0^\top|$ go to the completion detecting bus leading to the row inputs on y . Reordering them in a block $(S_2 d) S_1(d, x) S_0(d, y) \in \mathbb{H}$ with a function $S_2 : \mathcal{P}(\mathbb{N}^2)^2 \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ given by

$$S_2 = \lambda d. \lambda s. (\lambda j. (\lambda k. \mathbf{L}_k \langle \uparrow^k j, s \rangle \rangle) (|(d_1)_0^\top| + j) |(d_0)_1^\top|$$

puts all row inputs ahead of the column inputs. This result nevertheless places the outputs from x ahead of those from y , leaving a sparse decision wait with coordinates $S_3 d$ for $S_3 : \mathcal{P}(\mathbb{N}^2)^2 \rightarrow \mathcal{P}(\mathbb{N}^2)$ given by

$$S_3 = \lambda d. ((\mu \lambda b. \langle ((d_0 - d_1)_0^\top)^\circ b_0, b_1 \rangle) (d_0 - d_1)) \cup (\mu \lambda b. \langle (((d_1)_0^\top)^\circ b_0) + |(d_0)_0^\top| - 1, b_1 \rangle) d_1$$

whose rows therefore still need reordering to match the coordinates $c = \mathcal{U}_v^{-1} d$ even if the columns are already in order. In particular, the i -th row input terminal on this intermediate result needs to be driven by the u_i -th row input visible externally on the final result based on a list

$$u = \overset{\circ}{b} \langle (d_0 - d_1)_0^\top, (d_1)_0^\top \rangle$$

containing the row input indices associated with x followed by those of y . The transformation that reorders just the rows consistently without upsetting anything else would be

$$(\varphi_0 \langle (\overset{\circ}{b} \langle (d_0 - d_1)_0^\top, (d_1)_0^\top \rangle)^{-1}, \iota_{|(d_0)_1^\top|} \rangle) S_3 d : \mathbb{H} \rightarrow \mathbb{H}$$

based on [Equation 11.6](#), which is expressible more concisely as $(S_4 S_3) d$ in terms of a function

$$S_4 : (\mathcal{P}(\mathbb{N}^2)^2 \rightarrow \mathcal{P}(\mathbb{N}^2)) \rightarrow (\mathcal{P}(\mathbb{N}^2)^2 \rightarrow (\mathbb{H} \rightarrow \mathbb{H}))$$

defined as

$$S_4 = \lambda s. \lambda d. (\varphi_0 \langle (\overset{\circ}{b} \langle (d_0 - d_1)_0^\top, (d_1)_0^\top \rangle)^{-1}, \iota_{|(d_0)_1^\top|} \rangle) s d.$$

In terms of the foregoing functions, a combining form suitable for vertical spanning decompositions follows as

$$\Omega_v(d, x, y) = ((S_4 S_3) d) (S_2 d) S_1(d, x) S_0(d, y)$$

with the horizontal form not far off. To leverage the vertical form, we transform a triple

$$(d, x, y) \in \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \times \mathbb{H}$$

specifying a vertical decomposition to the analogous horizontal decomposition

$$S_5(d, x, y) \in \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \times \mathbb{H}$$

by reversing the coordinates in d to $(\mu \lambda b. b \circ \langle 1, 0 \rangle)^* d$ and rotating the building blocks x and y respectively to $r_0 x$ and $r_1 y$ using a list r of two rotational transformations

$$r = (\varphi_1 \langle 1, 0 \rangle)^* \eta d$$

by Equation 11.11 in a definition of $S_5 : \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \times \mathbb{H} \rightarrow \mathcal{P}(\mathbb{N}^2)^2 \times \mathbb{H} \times \mathbb{H}$ given by

$$S_5 = \lambda(d, x, y). (\lambda r. ((\mu \lambda b. b \circ \langle 1, 0 \rangle)^* d, r_0 x, r_1 y)) (\varphi_1 \langle 1, 0 \rangle)^* \eta d.$$

It would be meaningful to write $\Omega_v S_5(d, x, y)$ for a vertical analog to the given horizontal decomposition, but the coordinates of this result would be the reversals $(\mu \lambda b. b \circ \langle 1, 0 \rangle) c$ of the required coordinates $c = \mathcal{U}_h^{-1} d$ by Equation 11.19 due to these rotations. Correcting for this effect is straightforward by one further transformation $S_6 d : \mathbb{H} \rightarrow \mathbb{H}$ based on a function $S_6 : \mathcal{P}(\mathbb{N}^2)^2 \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ defined as

$$S_6 = \lambda d. (\lambda c. \varphi(\langle \langle 1, 0 \rangle, \iota_{(\sigma c)_1}, \iota_{(\sigma c)_0} \rangle, (\mu \lambda b. b \circ \langle 1, 0 \rangle) c)) \mathcal{U}_h^{-1} d$$

by Equation 11.5 in the specification of the horizontal combining form.

$$\Omega_h(d, x, y) = (S_6 d) \Omega_v S_5(d, x, y)$$

11.4.2 Enmeshed

When it is not possible or desirable to decompose a planar sparse decision wait vertically or horizontally, the enmeshed decomposition is another alternative. Figure 11.6 illustrates the ten thousand foot view. The idea is to clear a space in the lower right quadrant by shifting the more populous rows and columns upwards and to the left, and then to deal only with the three non-empty quadrants left over. A single large empty quadrant is conducive to a small routing network as shown in Figure 11.7, so unlike the quadrangular decomposition discussed in Section 10.3, this one sticks to a fixed number of quadrants.



Even so, the overhead due to the routing network in Figure 11.7 entails a trade-off. A degenerate combination $\Omega_h(c, \text{MDW } \sigma c)$ costs only an amount proportional to $(\prod \sigma c) - |c|$ more than the dense decision wait MDW σc with dimensions σc . For instances of $|c|$ close to $\prod \sigma c$ (that is, for nearly dense decision waits), this cost could be less than that of the routing network, to say nothing of performance, and maybe even less than the cost of the routing network discounted by the omission of the empty quadrant. An optimal decomposition therefore requires the right choice of a decomposition function $\mathcal{U}_e : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^2)^*$ to be used in Equation 11.14. As noted in the discussion of Equation 11.15, it can be chosen as any that satisfies

$$\forall c \in \mathcal{P}(\mathbb{N}^2). \mathcal{U}_e c \in (\nabla_e c) \cup \{\epsilon\} \tag{11.20}$$

for a function $\nabla_e : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^*)$ to be defined presently.

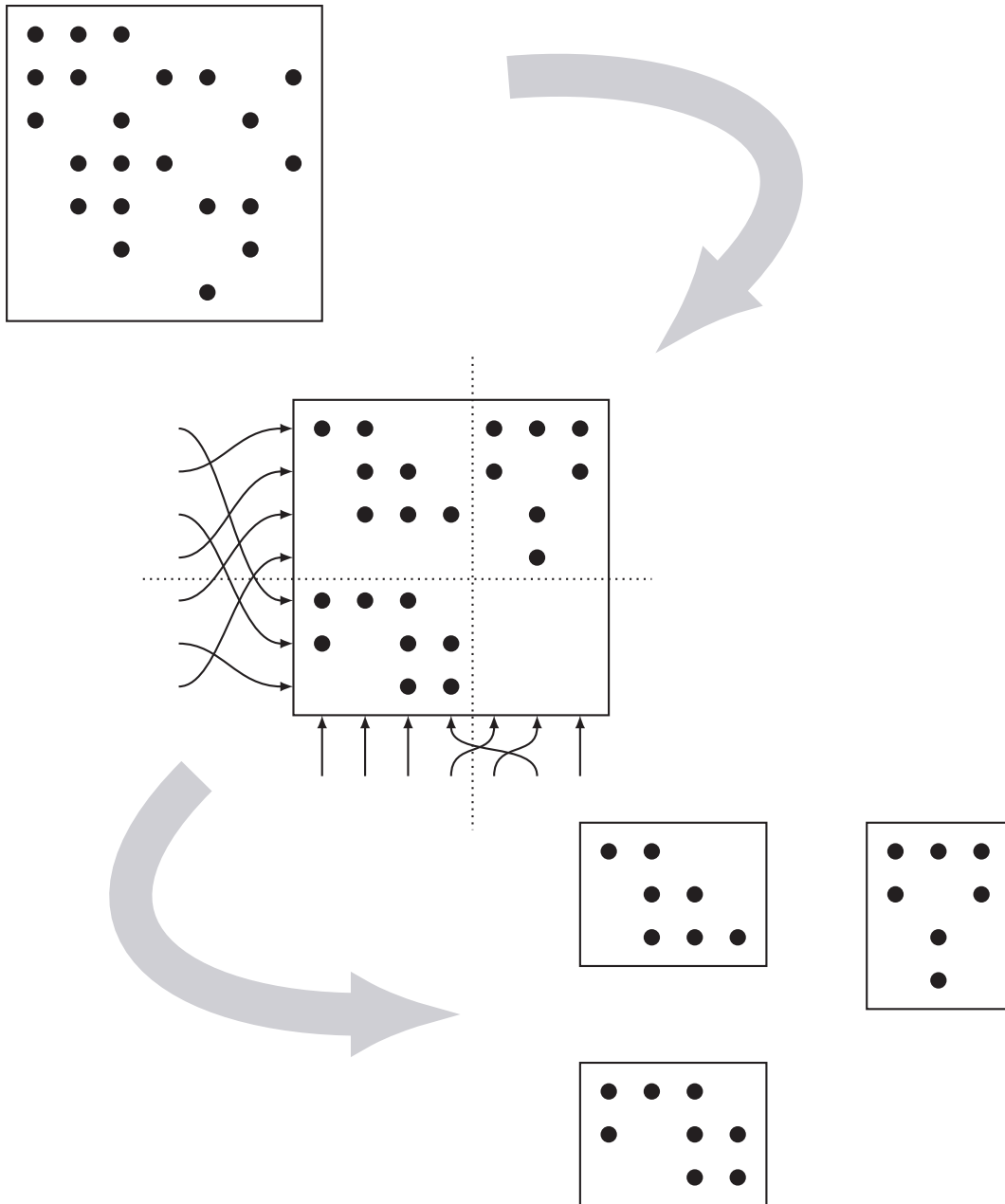


Figure 11.6: An inseparable enmeshed sparse decision wait (top) has its rows and columns permuted to clear a contiguous empty quadrant (center) so that the three remaining non-empty quadrants can be implemented individually and then glued together as in [Figure 11.7](#) (bottom).

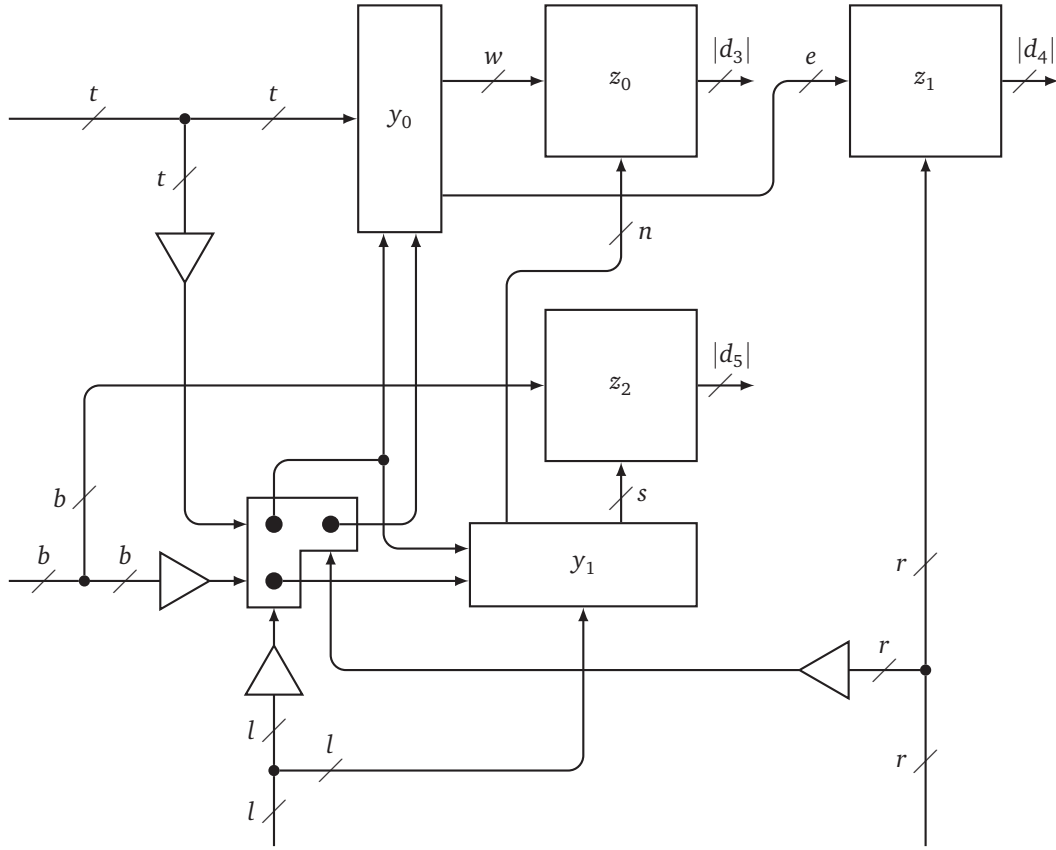


Figure 11.7: the enmeshed combination $\Omega_e(d, x, y, z)$, where $d = \mathcal{U}_e c$ is an enmeshed decomposition for some coordinates c , x is an LJOIN, the coordinates of y are $\eta \langle d_1, d_2 \rangle$, and the coordinates of z are $\eta \langle d_3, d_4, d_5 \rangle$ (cf. Figure 10.9)

Decomposition functions

To generalize from Figure 11.6, an enmeshed decomposition is determined by any non-empty, non-full, non-spanning set of rows containing points $u \subset c$ within a planar sparse decision wait specification $c \in \mathcal{P}(\mathbb{N}^2)$. Hence u must be a union of one or more members of $\mathcal{R}(v_0 c)$ by Equation 11.16 for which the proper subset relationships $u_0^T \subset c_0^T$ and $u_1^T \subset c_1^T$ both hold, although the latter implies the former because there are no empty columns. Perhaps not always an easy thing to compute efficiently, the set $s \in \mathcal{P}(\mathcal{P}(c))$ of candidates $u \in \mathcal{P}(c)$ may yield in some cases to a formula like $s = (v_4 v_0) c$, with

$$v_4 : (\mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)^*) \rightarrow (\mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)))$$

defined in terms of a fold over a list of the non-spanning rows rather than a naive enumeration of their power set.

$$v_4 = \lambda v. \lambda c. ((\mathcal{F}_{\{\emptyset\}} \lambda(h, z). z \cup \{u \in \mathcal{P}(\mathbb{N}^2) \mid u - h \in z \wedge u_1^T \subset c_1^T\}) v c) - \{\emptyset\} \tag{11.21}$$

Any known set $u \in (v_4 v_0) c$ then induces a list of three sets of points $d = \langle t - r, r, u \rangle$ corresponding to the top left, top right, and bottom left quadrants respectively in [Figure 11.6](#) with

$$t = c - ((u_0^\top)^1 \parallel (c_1^\top)^1)$$

covering the rows disjoint from u and

$$r = t - ((t_0^\top)^1 \parallel (u_1^\top)^1)$$

covering the columns disjoint from u . The set of all lists d of sets of points meeting these conditions is convenient to summarize as $(v_5 v_4 v_0) c$ in terms of

$$v_5 : (\mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*))) \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^3))$$

given by

$$v_5 = \lambda v. \lambda c. (\mu \lambda u. (\lambda t. (\lambda r. \langle t - r, r, u \rangle) (t - ((t_0^\top)^1 \parallel (u_1^\top)^1))) (c - ((u_0^\top)^1 \parallel (c_1^\top)^1))) v c$$

but perhaps even better to summarize as $v_3 (v_5 v_4 v_0) c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^3)$ by [Equation 11.17](#) to avoid empty quadrants that should be non-empty or other spurious results in cases of separable or higher dimensional coordinates c .

Auxiliary decomposition functions

The result derived up to this point indicates coordinates only for the three blocks $z = \langle z_0, z_1, z_2 \rangle$ in the enmeshed combination shown in [Figure 11.7](#) and not for the blocks $y = \langle y_0, y_1 \rangle$, but the complete decomposition should describe all five. The latter blocks are bicolumnar and bilateral sparse or dense decision waits respectively serving a similar purpose to the input routing stages of the quadrangular decision wait shown in [Figure 10.9](#). That is, the choice of a column input to y_0 determines whether the top t row inputs to the combination should be routed west to the row inputs on z_0 or east to the row inputs in z_1 . Similarly, the row input signal to y_1 selects either the north output bus to the column inputs on z_0 as the destination for a signal from the left l column bus lines, or south one to z_2 .

To serve their intended purpose, the two blocks y in [Figure 11.7](#) are restricted by any fixed choice of z having coordinates $d \in \mathcal{P}(\mathbb{N}^2)^3$ to uniquely determined coordinates $\eta u \in \mathcal{P}(\mathbb{N}^2)^2$ depending on d . Specifically, there must be one point of the form $\langle a, 0 \rangle$ in u_0 indicating the presence of an output in the left column of y_0 for each row of z_0 , and one point of the form $\langle b, 1 \rangle$ indicating the presence of an output in the right column of y_0 for each row of z_1 , implying coordinates $(\eta u)_0 \in \mathcal{P}(\mathbb{N}^2)$ for y_0 as the local renumbering of

$$u_0 = (((d_0)_0^\top)^1 \parallel \{0\}^1) \cup (((d_1)_0^\top)^1 \parallel \{1\}^1).$$

Similarly, there must be one output in the top row of y_1 for each column of z_0 , and one output in the bottom row of y_1 for each column of z_2 .

$$u_1 = (\{0\}^1 \parallel ((d_0)_1^\top)^1) \cup (\{1\}^1 \parallel ((d_2)_1^\top)^1)$$

The coordinates of y_i for either value of $i \in \{0, 1\}$ therefore would be a locally renumbered

$$u_i = (\{0\}^i \parallel ((d_0)_i^\top)^1 \parallel \{0\}^{1-i}) \cup (\{1\}^i \parallel ((d_{i+1})_i^\top)^1 \parallel \{1\}^{1-i}) = \bigcup_{j \in \{0,1\}} \{j\}^i \parallel ((d_{(i+1)j})_i^\top)^1 \parallel \{j\}^{\delta_i^j}.$$

To cover both cases, we may write $u = \dot{\mathcal{U}}_e d$ in terms of a function designated $\dot{\mathcal{U}}_e : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^2)^*$ to connote some sort of auxiliary enmeshed decomposition function. However, the expression for u_i above is undefined unless d is a list of at least three sets of two-dimensional points. It is helpful in other contexts to define $\dot{\mathcal{U}}_e d$ as an empty list ϵ when these conditions do not hold.

$$\dot{\mathcal{U}}_e = \lambda d. (\lambda k. \langle \epsilon, (\lambda i. \bigcup_{j \in \{0,1\}} \{j\}^i \parallel ((d_{(i+1)j})_i^\top)^1 \parallel \{j\}^{\delta_i^0})^* \iota_2 \rangle_k) \delta_{|\mathring{b}d|}^{|\bigcup \mathcal{R}(d)|} \delta_{\emptyset}^{\mathcal{U}_s^{-1}d} \delta_3^{|d|} \delta_2^{|\sigma \bigcup \mathcal{R}(d)|} \quad (11.22)$$

The condition $|\bigcup \mathcal{R}(d)| = |\mathring{b}d|$ that the terms of d are mutually disjoint and the condition $\mathcal{U}_s^{-1}d = \emptyset$ that they are not separable (Section 11.3.2) are not strictly necessary to ensure a well defined value of $\dot{\mathcal{U}}_e d$, but they simplify the definition of the inverse decomposition function \mathcal{U}_e^{-1} coming up shortly by precluding anomalous results when d does not represent an enmeshed decomposition.

Complete decomposition functions

At this point, we could identify the set of enmeshed decompositions for coordinates $c \in \mathcal{P}(\mathbb{N}^2)$ as something like

$$(\mu \lambda d. (\dot{\mathcal{U}}_e d) \parallel d) \nu_3 (\nu_5 \nu_4 \nu_0) c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^5)$$

to account for the coordinates d of the blocks z and the coordinates $\dot{\mathcal{U}}_e d$ of the blocks y in each possible decomposition (subject to local renumbering). However, the LJOIN x needed for the enmeshed combination is an explicit parameter to the combining form Ω_e in case there is more than one way to construct an LJOIN (which there is), so it is less troublesome in the long run to let its coordinates feature explicitly in the decomposition as well.

$$(\mu \lambda d. (\{0, 1\}^2 - \{1\}^2) : (\dot{\mathcal{U}}_e d) \parallel d) \nu_3 (\nu_5 \nu_4 \nu_0) c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^6)$$

Hence we define $\nabla_e : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^2)^6)$ in its most general form as

$$\nabla_e = (\mu \lambda d. (\{0, 1\}^2 - \{1\}^2) : (\dot{\mathcal{U}}_e d) \parallel d) \circ \nu_3 \circ (\nu_5 \nu_4 \nu_0). \quad (11.23)$$

Approximate decomposition functions

Similarly to spanning decompositions, there is at most one best choice of $\mathcal{U}_e c \in \nabla_e c$ for a specification c , and because ∇_e may difficult to compute efficiently, it is worthwhile to ask whether it can be approximated without missing too much. If one subscribes to the view that larger empty quadrants are always preferable to smaller ones, then there is nothing to lose by lumping the identically populated rows together in advance using

$$((\mu \lambda s. \bigcup s) \circ (\pi \lambda r. r_1^\top) \circ \lambda c. \mathcal{R}(\nu_0 c))^{-1}$$

in place of ν_0 in Equation 11.23. In many cases this optimization implies a shorter list to fold in Equation 11.21 (a likely computational bottleneck), with less work to do on each round. A definition of an approximate version of ∇_e incorporating this optimization would be as follows.

$$\nabla_e = (\mu \lambda d. (\{0, 1\}^2 - \{1\}^2) : (\dot{\mathcal{U}}_e d) \parallel d) \circ \nu_3 \circ (\nu_5 \nu_4 ((\mu \lambda s. \bigcup s) \circ (\pi \lambda r. r_1^\top) \circ \lambda c. \mathcal{R}(\nu_0 c))^{-1})$$

Inverse decomposition functions

An inverse function for enmeshed decomposition should be able to take any form of sparse decision wait decomposition $d \in \mathcal{P}(\mathbb{N}^*)^*$, determine whether it corresponds to an enmeshed decomposition (as opposed to some other form), and recover the coordinates $c = \mathcal{U}_e^{-1} d \in \mathcal{P}(\mathbb{N}^2)$ if it does. Although the choice of \mathcal{U}_e may vary, a fixed inverse $\mathcal{U}_e^{-1} : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^2)$ is adequate for any choice of \mathcal{U}_e satisfying Equation 11.20. It need only map d either to the union $\bigcup \mathcal{R}(d \ll 3)$ of its last three terms or to the empty set \emptyset depending on certain conditions.

The obvious conditions are the that length $|d|$ is 6, the initial term $d_0 = \{0, 1\}^2 - \{1\}^2$ contains the coordinates of an LJOIN, and the next two terms $\langle d_1, d_2 \rangle$ match the value $\mathcal{U}_e(d \ll 3)$ of the last three by Equation 11.22. This last condition would imply a planar non-separable list of sets of points $d \ll 3$ occupying a contiguous sequence of rows and columns.

To nail it down more completely, we can require the upper right and lower left quadrants to have no rows or columns in common,

$$((d_4)_0^T \cap (d_5)_0^T) \cup ((d_4)_1^T \cap (d_5)_1^T) = \bigcap_{j=4}^5 (d_j)_0^T \cup \bigcap_{j=4}^5 (d_j)_1^T = \bigcup_{i=0}^1 \bigcap_{j=4}^5 (d_j)_i^T = \emptyset$$

and the upper left quadrant to have no columns in common with the upper right

$$(d_3)_1^T \cap (d_4)_1^T = \bigcap_{k \in \{3,4\}} (d_k)_1^T = \emptyset$$

or rows in common with the lower left

$$(d_3)_0^T \cap (d_5)_0^T = \bigcap_{k \in \{3,5\}} (d_k)_0^T = \emptyset$$

all of which can be summarized as

$$t = \bigcup_{i=0}^1 \bigcap_{j=4}^5 (d_j)_i^T \cup \bigcap_{k \in \{3, i+4\}} (d_k)_i^T = \emptyset$$

in a definition for \mathcal{U}_e^{-1} of the following form.

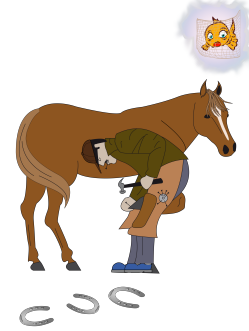
$$\mathcal{U}_e^{-1} = \lambda d. (\lambda s. \langle \emptyset, (\lambda t. \langle \emptyset, \bigcup \mathcal{R}(d \ll 3) \rangle_{\delta_t^e}) \bigcup_{i=0}^1 \bigcap_{j=4}^5 (d_j)_i^T \cup \bigcap_{k \in \{3, i+4\}} (d_k)_i^T \rangle_s) \delta_6^{|d|} \delta_{d+1}^{\langle \{0,1\}^2 - \{1\}^2 \rangle} \delta_{(d \ll 1)+2}^{\mathcal{U}_e(d \ll 3)}$$

Combining form

Now that we know how to decompose the coordinates $c \in \mathcal{P}(\mathbb{N}^2)$ of a desired planar sparse decision wait into those of the building blocks shown in Figure 11.7, which could be implemented separately more easily, all we need is a way of putting them together to finish the job. This task is best understood in terms of a combining form

$$\Omega_e : \mathcal{P}(\mathbb{N}^2)^6 \times \mathbb{H} \times \mathbb{H}^2 \times \mathbb{H}^3 \rightarrow \mathbb{H}$$

taking a decomposition $d = \mathcal{U}_e c \in \mathcal{P}(\mathbb{N}^2)^6$ of the desired coordinates c , an LJOIN $x \in \mathbb{H}$, two known routing stages $y \in \mathbb{H}^2$, and three sparse decision waits $z \in \mathbb{H}^3$ for the three non-empty quadrants, also



presumed given, to a result $\Omega_e(d, x, y, z) \in \mathbb{H}$ with coordinates c . The coordinates of the building blocks are assumed to match d after renumbering, with the coordinates of the routing stages y equal to $\eta \langle d_1, d_2 \rangle$ and z equal to $\eta \langle d_3, d_4, d_5 \rangle$, whereas anything else there is to know about c can be inferred without ambiguity from $c = \mathcal{U}_e^{-1} d$. These conditions are met by hypothesis in Equation 11.14.

Lengthy but mostly straightforward in its derivation, the enmeshed combinator takes the form of a front end containing x and y connected to a back end $(\mathcal{F} \mathbf{R}) z$ by a central permutation network depending on d , with a further input permutation network on the front end and an output permutation network on the back end.

$$\Omega_e(d, x, y, z) = F_4 F_1 d \times (F_3 F_2 F_0(d, x, y)) F_1 d \xrightarrow{F_5 d} (\mathcal{F} \mathbf{R}) z \times F_6 d \quad (11.24)$$

The rest of this section specifies the functions F_0 through F_6 needed to make it work.

Front end To jump into the thick of it, the LJOIN x shown Figure 11.7 needs a FORK connected to its first output, as in $\mathbf{Z}\mathbf{R}(x, \text{FORK})$, with one output from the FORK to the first column or row input to each of y_0 and y_1 respectively, and each of remaining outputs from x connected to the other column or row input on one of y_0 or y_1 . Shuffling the outputs from the former in a block $\mathbf{Z}\mathbf{R}(x, \text{FORK}) \uparrow_2^1$ and the inputs to the latter in a block

$$\mathbf{R}(\varphi_1 \langle 1, 0 \rangle (\eta \langle d_1, d_2 \rangle)_0 y_0, y_1 \uparrow 2) \uparrow 2$$

with y_0 transformed by Equation 11.11 from bicolunar to bilateral, then combining both in parallel

$$\mathbf{R}(\mathbf{Z}\mathbf{R}(x, \text{FORK}) \uparrow_2^1, \mathbf{R}(\varphi_1 \langle 1, 0 \rangle (\eta \langle d_1, d_2 \rangle)_0 y_0, y_1 \uparrow 2) \uparrow 2)$$

enables all four connections from the FORK and LJOIN to the routing stages y in a block

$$\mathbf{Z}^4 \mathbf{R}(\mathbf{Z}\mathbf{R}(x, \text{FORK}) \uparrow_2^1, \mathbf{R}(\varphi_1 \langle 1, 0 \rangle (\eta \langle d_1, d_2 \rangle)_0 y_0, y_1 \uparrow 2) \uparrow 2)$$

with the inputs to the LJOIN moved to the last positions in

$$(\mathbf{Z}^4 \mathbf{R}(\mathbf{Z}\mathbf{R}(x, \text{FORK}) \uparrow_2^1, \mathbf{R}(\varphi_1 \langle 1, 0 \rangle (\eta \langle d_1, d_2 \rangle)_0 y_0, y_1 \uparrow 2) \uparrow 2)) \uparrow 4$$

hereafter abbreviated $F_0(d, x, y) \in \mathbb{H}$ with $F_0 : \mathcal{P}(\mathbb{N}^2)^6 \times \mathbb{H} \times \mathbb{H}^2 \rightarrow \mathbb{H}$ given by

$$F_0 = \lambda(d, x, y). (\mathbf{Z}^4 \mathbf{R}(\mathbf{Z}\mathbf{R}(x, \text{FORK}) \uparrow_2^1, \mathbf{R}(\varphi_1 \langle 1, 0 \rangle (\eta \langle d_1, d_2 \rangle)_0 y_0, y_1 \uparrow 2) \uparrow 2)) \uparrow 4.$$

Hence $F_0(d, x, y)$ exposes the outputs corresponding to the buses whose widths are labeled w, e, n , and s in that order in Figure 11.7 (mnemonic for “west”, “east”, “north” and “south”). On the input side, the t row inputs to y_0 are first, followed by the l column inputs to y_1 (for “top” and “left”), followed by the four LJOIN inputs meant for the four MERGE outputs from the completion detecting buses with widths t, b (for “bottom”), l , and r (for “right”) in that order.

Focusing next on the bus widths t, b, l , and r , we note that they follow from the cardinalities respectively of the list of four sets of coordinates $F_1 d \in \mathcal{P}(\mathbb{N})^4$ for $F_1 : \mathcal{P}(\mathbb{N}^2)^6 \rightarrow \mathcal{P}(\mathbb{N})^4$ given by

$$F_1 = \lambda d. \langle (d_1)_0^T, (d_5)_0^T, (d_2)_1^T, (d_4)_1^T \rangle$$

these being the sets of row inputs to y_0 , row inputs to z_2 , column inputs to y_1 , and column inputs to z_1 in that order numbered relative to the original specification c . We could therefore make a list

$$(\lambda k. \mathbf{L}_{|k|} \langle \text{FORK}^{|k|} \uparrow_2^1, \text{MERGE } |k| \rangle) * F_1 d \in \mathbb{H}^4$$

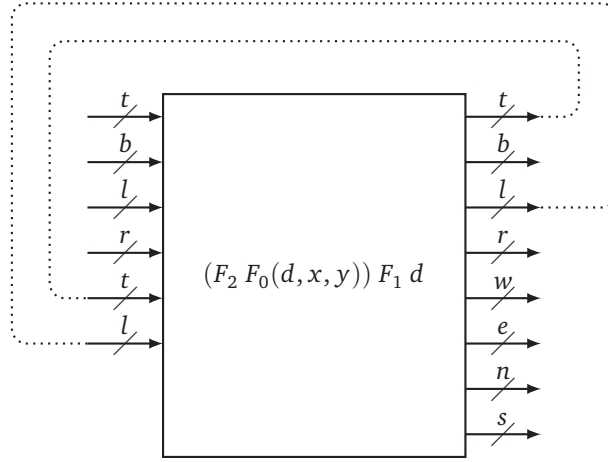


Figure 11.8: A partial description of the front end can be transformed to $(F_3 F_2 F_0(d, x, y)) F_1 d$ by making the indicated connections.

of the four completion detecting buses folded into a single network

$$(\mathcal{F} \lambda(h, u). \mathbf{R}(h \downarrow 1, u) \uparrow 1) (\lambda k. \mathbf{L}_{|k|} \langle \text{FORK}^{|k|} \uparrow_2^1, \text{MERGE } |k| \rangle) * F_1 d \in \mathbb{H}$$

whose inputs would be for buses with widths $t, b, l,$ and r in that order, and whose outputs would be from these buses in the same order followed in the last four positions by their respective completion detection signals in the reverse order. Connecting the rest of the routing network $F_0(d, x, y)$ to the completion detecting buses takes only a rotation and four more connections

$$\mathbf{Z}^4 \mathbf{R}(((\mathcal{F} \lambda(h, u). \mathbf{R}(h \downarrow 1, u) \uparrow 1) (\lambda k. \mathbf{L}_{|k|} \langle \text{FORK}^{|k|} \uparrow_2^1, \text{MERGE } |k| \rangle) * F_1 d) \downarrow 4, F_0(d, x, y))$$

enabling a description $(F_2 F_0(d, x, y)) F_1 d \in \mathbb{H}$ with $F_2 : \mathbb{H} \rightarrow (\mathcal{P}(\mathbb{N})^4 \rightarrow \mathbb{H})$ given by

$$F_2 = \lambda f. \lambda v. \mathbf{Z}^4 \mathbf{R}(((\mathcal{F} \lambda(h, u). \mathbf{R}(h \downarrow 1, u) \uparrow 1) (\lambda k. \mathbf{L}_{|k|} \langle \text{FORK}^{|k|} \uparrow_2^1, \text{MERGE } |k| \rangle) * v) \downarrow 4, f)$$

and having input and output buses whose ordering and widths are depicted in Figure 11.8.

The figure also suggests that the next connections worth making are from the output buses having widths t and l to the input buses of those widths. These connections are achievable by a parallel combination with a bus of width $t + l$ in a block

$$\mathbf{R}((F_2 F_0(d, x, y)) F_1 d, \mathbf{l}^{t+l})$$

followed by a connection of the top t outputs from the existing block to the parallel bus, then by a roll of the next b outputs out of the way, and then by a connection of the next l outputs to the parallel bus in

$$\mathbf{Z}^l((\mathbf{Z}^t \mathbf{R}((F_2 F_0(d, x, y)) F_1 d, \mathbf{l}^{t+l})) \uparrow b)$$

which leaves buses of widths $l, t,$ and b in the last output positions, so by rolling them to the top and connecting two of them to the last two input buses

$$\mathbf{Z}^{t+l}((\mathbf{Z}^l((\mathbf{Z}^t \mathbf{R}((F_2 F_0(d, x, y)) F_1 d, \mathbf{l}^{t+l})) \uparrow b)) \downarrow t + b + l)$$

we have a result resembling [Figure 11.8](#) with the connections made. Denote this revised version of the front end $(F_3 F_2 F_0(d, x, y)) F_1 d$ in terms of a function $F_3 : (\mathcal{P}(\mathbb{N})^4 \rightarrow \mathbb{H}) \rightarrow (\mathcal{P}(\mathbb{N})^4 \rightarrow \mathbb{H})$ defined as

$$F_3 = \lambda f. \lambda v. (\lambda(t, b, l). \mathbf{Z}^{t+l}((\mathbf{Z}^l((\mathbf{Z}^t \mathbf{R}(f v, l^{t+l})) \uparrow b)) \downarrow t + b + l)) (|v_0|, |v_1|, |v_2|).$$

Input permutation network The inputs shown in [Figure 11.8](#) are probably in the wrong order for the original specification $c = \mathcal{U}_e^{-1} d$, but fortunately are easy to correct. Letting

$$\langle t, b, l, r \rangle = F_1 d \in \mathcal{P}(\mathbb{N})^4$$

refer momentarily to the list of sets of externally visible line numbers associated with each of the internal input buses (that is, not just their widths), we construct permutations for the rows and columns separately

$$p = \mathring{b}^* \langle \langle t, b \rangle, \langle l, r \rangle \rangle \in \mathbb{N}^{*2}$$

whose inverses combine to specify the input permutation network

$$p_0^{-1} \parallel (\lambda i. |p_0| + i)^* p_1^{-1} \in \mathbb{N}^*$$

by way of a concatenation of the row input permutation with a list of column terminal numbers derived from the column input permutation offset by the number of rows. This list is more easily expressible as $F_4 F_1 d$ with $F_4 : \mathcal{P}(\mathbb{N})^4 \rightarrow \mathbb{N}^*$ defined as

$$F_4 = \lambda \langle t, b, l, r \rangle. (\lambda p. p_0^{-1} \parallel (\lambda i. |p_0| + i)^* p_1^{-1}) \mathring{b}^* \langle \langle t, b \rangle, \langle l, r \rangle \rangle$$

so that at this point we have the whole front end

$$F_4 F_1 d \times (F_3 F_2 F_0(d, x, y)) F_1 d \in \mathbb{H}$$

including the input permutation network.

Central permutation network Having derived the front end as shown above and assuming a parallel combination of the blocks z for the back end in [Equation 11.24](#), we are obliged to implement the central permutation network as shown in [Figure 11.9](#) because the input buses to the back end are ordered differently from their sources on the front end. The bus widths ordered by their back end destinations are expressible easily enough as

$$u = \langle w, n, e, r, b, s \rangle = \mathring{b} (\sigma^* \eta d \ll 3) \in \mathbb{N}^6$$

in terms of the dimensions $\sigma^* \eta d \ll 3 \in (\mathbb{N}^2)^3$ of the back end blocks z_0, z_1 , and z_2 as given by the decomposition d , but the first b lines out of the front end are destined for the row inputs on z_2 , the next r for the column inputs on z_1 , etc., with no obvious pattern. To take an *ad hoc* approach, we can first form the list

$$v = (\lambda i. \iota_{u_i}^{\sum(u \uparrow i)})^* \iota_{|u_i|} \in (\mathbb{N}^*)^6$$

of the six lists of input terminal numbers relative to the parallel combination $(\mathcal{F} \mathbf{R}) z$ for which the length u_i of each list v_i is that of the i -th of six rows and columns of $(\mathcal{F} \mathbf{R}) z$ and the terms in each v_i are offset by the cumulative length $\sum(u \uparrow i)$ of its predecessors. By inspection of the [Figure 11.9](#), permuting the whole list v by $\langle 4, 3, 0, 2, 1, 5 \rangle$ and flattening the result should lead to the necessary permutation $F_5 d$ overall for $F_5 : \mathcal{P}(\mathbb{N})^6 \rightarrow \mathbb{N}^6$ defined as

$$F_5 = \lambda d. \mathring{b} (((\lambda u. (\lambda i. \iota_{u_i}^{\sum(u \uparrow i)})^* \iota_{|u_i|}) \mathring{b} (\sigma^* \eta d \ll 3)) \circ \langle 4, 3, 0, 2, 1, 5 \rangle).$$

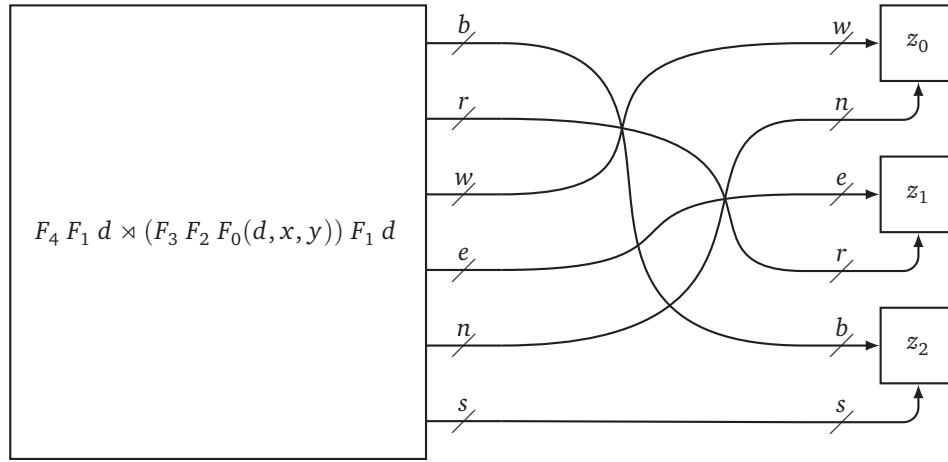


Figure 11.9: central permutation network $F_5 d$, with $\langle w, n, e, r, b, s \rangle = b (\sigma^* \eta d \ll 3)$ (cf. Figure 11.7 and Figure 11.8)

Output permutation network The output lines from the back end blocks z_0 , z_1 , and z_2 are in a different order from what is required to implement the original specification $c = \mathcal{U}_e^{-1} d$, with all outputs from z_0 preceding those of z_1 and z_1 preceding those of z_2 , but an output permutation network can compensate by reordering them.

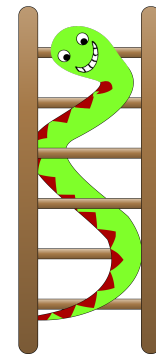
The way the output lines from z need to be interwoven is readily apparent from the three lists d_3 , d_4 , and d_5 indicated by the decomposition, which correspond to the coordinates of z but are numbered globally relative to c . Although c is not separable, the three lists $d \ll 3$ have mutually disjoint ranges whose union is c as in a separable decomposition, so the same idea for an output permutation described on page 326 works here as well. That is, a list $(\mathcal{U}_e^{-1} d)^{\circ-1}$ of the points in c ordered by the positions of their corresponding output terminals induces a list

$$(\mathring{b}(d \ll 3))^{-1*} (\mathcal{U}_e^{-1} d)^{\circ-1}$$

of the positions of the output terminals on $(\mathcal{F} \mathbf{R}) z$ driving each one in the corresponding order, which completely describes the output permutation as $F_6 d$ for $F_6 : \mathcal{P}(\mathbb{N}^2)^6 \rightarrow \mathbb{N}^*$ defined as follows.

$$F_6 = \lambda d. (\mathring{b}(d \ll 3))^{-1*} (\mathcal{U}_e^{-1} d)^{\circ-1}$$

This result concludes not only the specification of Equation 11.24 for enmeshed combination, but the sparse decision wait generating function in Equation 11.14.



11.5 Multidimensional sparse decision waits

Nothing stops sparse decision waits from having more than two coordinates per point, which would imply sparse versions of the multidimensional decision waits developed in Section 10.4. Because

the output arities of the latter grow cubically or more with their input arities, a multidimensional sparse alternative is all the more economical where applicable. However, the sparse decision wait generating function defined by Equation 11.14 does not take full advantage of this possibility unless the coordinate specification is separable, which is probably rare in practice. A generalization catering more effectively to multidimensional sparse decision waits by adapting the dendriform (Section 10.4.1) and crossbar (Section 10.4.2) decompositions to them might look something like this.

$$\text{MSDW}(c) = \begin{cases} (\lambda d. \Omega_s(d, \text{MSDW}^*(\eta d)) \mathcal{U}_s c & \text{if } \mathcal{U}_s c \neq \epsilon \\ (\lambda d. \Omega_d(d, \text{MSDW}^*(\eta d \uparrow |d| - 1), \text{MSDW } d_{|d|-1})) \mathcal{U}_d c & \text{if } \mathcal{U}_d c \neq \epsilon \\ (\lambda d. (\lambda n. \Omega_c(d, \text{MSDW}^*(\eta d \uparrow n), \text{MSDW}^*(\eta d \ll n)) |(d_{|d|-1})^\top|) \mathcal{U}_c c & \text{if } \mathcal{U}_c c \neq \epsilon \\ \text{SDW } c & \text{otherwise} \end{cases}$$

Using the same functions \mathcal{U}_s and Ω_s as before for separable coordinates, and devolving to one of the previously defined sparse decision wait generating functions SDW in planar and degenerate cases, this upgraded version needs only the new decomposition functions \mathcal{U}_d and \mathcal{U}_c for dendriform and crossbar decompositions respectively, and their associated combining forms Ω_d and Ω_c . More about dendriform sparse decision waits follows presently in Section 11.5.1, with a discussion of crossbar sparse decision waits in Section 11.5.2.

11.5.1 Dendriform

The dendriform sparse decision wait follows the same pattern as Figure 10.16, with at least two leaf blocks driving a central root, but any of the building blocks can be either a sparse or a dense decision wait, so the bus widths are not necessarily products of the dimensions. Another difference is that the coordinates of the root are not uniquely determined by those of the leaves (even though its dimensions are), so any valid decomposition function $\mathcal{U}_d : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)^*$ needs to specify a result $d = \mathcal{U}_d c$ with one term d_i for each leaf x_i , and an additional term $d_{|d|-1}$ for the root y , by convention the last in the list. The good news is that the combining form

$$\Omega_d : \mathcal{P}(\mathbb{N}^*)^* \times \mathbb{H}^* \times \mathbb{H} \rightarrow \mathbb{H}$$

under these assumptions is quite simple

$$\Omega_d(d, x, y) = (\lambda k. \mathbf{C}_k \langle (\mathcal{F} \mathbf{R}) x, y \rangle) \sum \sigma d_{|d|-1} \quad (11.25)$$

requiring only one big bus from a parallel combination $(\mathcal{F} \mathbf{R}) x$ of the leaves x to the root y , with no twists or turns, whose width $k = \sum \sigma d_{|d|-1}$ is the total number of inputs to the root. With that settled, the rest of this section focuses only on dendriform decomposition functions and inverse decomposition functions.

Decomposition functions

A dendriform decomposition function \mathcal{U}_d chosen freely from the family of functions satisfying

$$\forall c \in \mathcal{P}(\mathbb{N}^*). \mathcal{U}_d c \in (\nabla_d c) \cup \{\epsilon\} \quad (11.26)$$

allows unlimited discretion regarding the use of dendriform decompositions in the context of a sparse decision wait generating function where there may be other alternatives, so its choice is left open subject only to the definition of $\nabla_d : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*)$ proposed presently.

One way to enumerate the possible dendriform decompositions of a specification $c \in \mathcal{P}(\mathbb{N}^*)$ is to start by enumerating all lists $s \in \mathbb{N}^{**}$ whose concatenation $\flat s = \sigma c$ coincides with the dimensions inferred from c . Then $|s|$ would be the number of leaves, s_i would be the dimensions of the i -th leaf, and presumably their coordinates and those of the root would be constrained enough by c somehow to determine a decomposition. Working backwards from σc to cut it into consecutive proper sublists amounts to selecting a set of places to cut it

$$p \in \mathcal{P}(\mathcal{R}(\iota_{n-1}^1))$$

inducing an ordered list $t = p^{\circ-1}$, where $n = |\sigma c|$ is the number of dimensions,¹ such that we envision each sublist s_i to contain terms $(\sigma c)_{t_i}$ through $(\sigma c)_{t_{i+1}-1}$, at least if we “pad” t by defining it as a member of $\nu_6 c$ for

$$\nu_6 : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)$$

given by

$$\nu_6 = \lambda c. (\lambda n. ((\mu \lambda p. 0 : p^{\circ-1} \parallel \langle n \rangle) \mathcal{P}(\mathcal{R}(\iota_{n-1}^1))) - (\mathbb{N}^2 \cup \mathbb{N}^{n+1})) |\sigma c|$$

where we also take the opportunity to exclude decompositions with only one leaf and with nothing but one-dimensional leaves.

We can now get a step further by pondering a set of lists of lists of coordinates derived from the specification $c \in \mathcal{P}(\mathbb{N}^*)$ and a particular choice of $t \in \nu_6 c$ as

$$r = (\mu \lambda b. (\lambda i. (b \ll t_i) \upharpoonright t_{i+1} - t_i)^* \iota_{|t|-1}) c \in \mathcal{P}(\mathbb{N}^{**})$$

from which the first part of the decomposition d describing leaves follows trivially as $r^\top \in \mathcal{P}(\mathbb{N}^*)^*$, and the last term of d describing the root is soon to be at hand. Each member $e \in r$ is a list of lists of coordinates wherein the i -th term e_i is a list of coordinates associated with the i -th leaf. The lexicographic ordinal of e_i relative to the set r_i^\top of i -th terms of all members of r identifies an output terminal number from the i -th leaf, hence the i -th coordinate of one point of the root. The whole point would be given by

$$(\lambda i. r_i^{\top \circ} e_i)^* \iota_{|e|}$$

and therefore the whole set of points in the root by $(\mu \lambda e. (\lambda i. r_i^{\top \circ} e_i)^* \iota_{|e|}) r$. Before it becomes any more of a mouthful, let the dendriform decomposition of coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ induced by a particular $t \in \nu_6 c$ be denoted $(\nu_7 t) c \in \mathcal{P}(\mathbb{N}^*)^*$ with

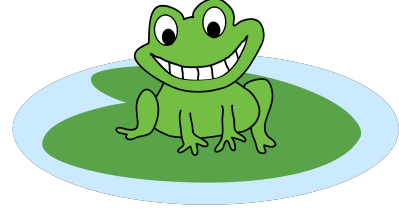
$$\nu_7 : \mathbb{N}^* \rightarrow (\mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)^*)$$

defined as

$$\nu_7 = \lambda t. (\lambda r. r^\top \parallel \langle (\mu \lambda e. (\lambda i. r_i^{\top \circ} e_i)^* \iota_{|e|}) r \rangle) \circ \mu \lambda b. (\lambda i. (b \ll t_i) \upharpoonright t_{i+1} - t_i)^* \iota_{|t|-1}.$$

To wrap up the definition of ∇_d , an expression encompassing the decompositions due to all possible values of $t \in \nu_6 c$ would be the map $(\mu \nu_7) \nu_6 c$, but technically this result constitutes a

¹There is no escape from the exponentially large number of sets p relative to n , so a resource constrained approach might limit choices of p to low cardinalities. The worst that would happen would be dendriform decompositions limited to binary or ternary trees, etc..



set of functions each needing to be applied to c to yield a decomposition d , so the actual set of decompositions is expressible as $(\mu \lambda t. (v_7 t) c) v_6 c$. Furthermore, there is never a good reason to use a dendriform decomposition when a separable decomposition is possible, so it is worth suppressing dendriform decompositions by hand if c is separable, meaning $\mathcal{U}_s c \neq \epsilon$. On a related note, there is no guarantee that a dendriform decomposition d for coordinates c by these criteria might not coincide with the separable decomposition of some other coordinates. For reasons to become clearer in [Section 11.6](#), it is desirable to avoid ambiguity in this unlikely event, so we must forgo members of $(\mu \mathcal{U}_s) \mathcal{P}(\mathcal{P}(\mathbb{N}^*))$ even if they are otherwise valid (in practice, by deleting any d satisfying $\mathcal{U}_s^{-1} d \neq \emptyset$ from the set of results) to arrive at the following definition overall.

$$\nabla_d = \lambda c. (\lambda i. \langle \emptyset, (\mu \lambda t. (v_7 t) c) v_6 c \rangle_i) \delta_\epsilon^{\mathcal{U}_s c} - (\mu \mathcal{U}_s) \mathcal{P}(\mathcal{P}(\mathbb{N}^*)) \quad (11.27)$$

Inverse decomposition functions

Inferring coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ from a non-empty decomposition $d \in \mathcal{P}(\mathbb{N}^*)^*$ whenever d is of the form $\mathcal{U}_d c$ for some \mathcal{U}_d satisfying [Equation 11.26](#) would proceed by inspecting each point $b \in \mathbb{N}^*$ in the set of points $d_{|d|-1}$ describing the root. Its i -th term $b_i \in \mathbb{N}$ is associated with a point

$$d_i^{\circ-1} b_i \in \mathbb{N}^*$$

in the i -th leaf, which is described by the i -th term d_i of the decomposition d . A concatenation of these points

$$b (\lambda i. d_i^{\circ-1} b_i)^* \iota_{|b|}$$

recovers a member of c , thus given in its entirety as $\dot{\mathcal{U}}_d^{-1} d$ for $\dot{\mathcal{U}}_d^{-1} : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^*)$ defined by

$$\dot{\mathcal{U}}_d^{-1} = \lambda d. (\lambda e. \langle \emptyset, (\mu \lambda b. b (\lambda i. d_i^{\circ-1} b_i)^* \iota_{|b|}) d_{|d|-1} \rangle_{\delta_\emptyset}) \mathcal{U}_s^{-1} d$$

and taking the precaution not to infer anything in the unlikely event of d also describing a separable decomposition, in keeping with [Equation 11.27](#).

Maybe not cautious enough, this result is either undefined or misleading if d is not a dendriform decomposition, because it relies on the conditions of d having more than two terms, the dimensionality of the last term matching the number of preceding terms, and the dimensionality of each preceding term d_i matching the i -th dimension of the last term. A generalization of $\dot{\mathcal{U}}_d^{-1}$ defined as

$$\mathcal{U}_d^{-1} = (\lambda j. \langle \emptyset, (\lambda k. \langle \emptyset, \dot{\mathcal{U}}_d^{-1} d \rangle_k) (\lambda m. (\lambda l. \langle 0, \prod (\lambda i. \delta_{|d_i|}^{(\sigma^{d_m})_i})^* \iota_m \rangle_l) \delta_m^{|\sigma^{d_m}|}) |d| - 1 \rangle_j) \delta_\epsilon^{\iota_3 \uparrow \{|d|\}}$$

satisfies $\mathcal{U}_d^{-1} d = \emptyset$ when any of these three conditions fails but is otherwise equal to $\dot{\mathcal{U}}_d^{-1} d$.

11.5.2 Crossbar

The crossbar decomposition for sparse decision waits follows the same pattern shown in [Figure 10.17](#), with a combination of front end blocks in parallel each connected to every one of a combination of back end blocks in parallel. It is also similar in that the front end blocks share most of their inputs through a FORK network except for those along their respective last dimensional axes. Furthermore, the dimensionality of each front end block matches the number of back end blocks, all of which have the same number of dimensions, and the sum of the front and back dimensionalities is the

successor of that of the whole, as in the dense crossbar decomposition. The theory of operation is also broadly similar to the description on page 301, but the rest of the details differ.

One difference is that the back end blocks are not necessarily identical beyond their dimensions, but may vary in their coordinates as the front also must. A valid decomposition function

$$\mathcal{U}_\zeta : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^*)^*$$

needs to provide a full account of all coordinates in a list $d = \mathcal{U}_\zeta c \in \mathcal{P}(\mathbb{N}^*)^*$ with one term d_i for each block, front end blocks first by convention. As usual, the choice of a decomposition function is flexible subject only to

$$\forall c \in \mathcal{P}(\mathbb{N}^*). \mathcal{U}_\zeta c \in (\nabla_\zeta c) \cup \{\epsilon\} \quad (11.28)$$

for the function $\nabla_\zeta : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*)$ to be defined shortly.

Another difference is that the number of back end blocks in a crossbar decomposition $d = \mathcal{U}_\zeta c$ for a sparse decision wait with coordinates c is not fully determined by the dimensions σc , being given instead by

$$|(\mu \lambda b. b \mid k) c|$$

the number of unique prefixes of length k of points $b \in c$, where $k = |(d_0)^\top| - 1$ is the predecessor of the front end dimensionality. The number of back end blocks attains a maximum of $\prod((\sigma c) \mid k)$ only when the decision wait is sufficiently dense.

While there are invariably $|(d_{|d|-1})^\top| = |c^\top| - k$ front end blocks each having $|\mu \lambda b. b \mid k) c|$ dimensions as implied above, the length along the j -th dimension may vary from one front end block to another. This effect is due to the need for a dedicated line within j -th bus from the i -th front end block to drive every i -th dimensional input of the j -th back end block, which is not generally constant, but instead something more like

$$|(((\pi \lambda b. b \mid k) c)^{\circ-1})_j)_{k+i}^\top|$$

if we had to put a number to it. Further analysis along these lines leads to the requisite decomposition functions, inverse decomposition functions, and combining form derived in the rest of this section to complete the definition of a multidimensional sparse decision wait generating function.

Decomposition functions

To continue the discussion above, let k range from 1 to $|c^\top| - 2$ inclusive for coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ with $|c^\top| > 2$. Then the crossbar decomposition determined by any fixed k features a list of back end blocks with coordinates obtained as a partition of the points in c by equality of their first k terms

$$((\pi \lambda b. b \mid k) c)^{\circ-1}$$

and the prefixes discarded once the list is made.

$$(\mu \lambda b. b \ll k)^* ((\pi \lambda b. b \mid k) c)^{\circ-1}$$

On the front end, the i -th block of $|c^\top| - k$ contains a point of the form

$$(b \mid k) \parallel \langle b_{k+i} \rangle$$

for each point $b \in c$, reflecting the shared k input buses among all front end blocks and the additional input in the $(k + i)$ -th dimension specific to that block, but the number of distinct points of this

form is generally less than $|c|$ because of the omitted coordinates. The whole i -th front end block specification is therefore

$$(\mu \lambda b. (b \upharpoonright k) \parallel \langle b_{k+i} \rangle) c$$

so the whole front end specification is

$$(\lambda i. (\mu \lambda b. (b \upharpoonright k) \parallel \langle b_{k+i} \rangle) c)^* \iota_{|c^\top| - k}$$

and a concatenation of the front and back end specifications makes a whole decomposition

$$((\lambda i. (\mu \lambda b. (b \upharpoonright k) \parallel \langle b_{k+i} \rangle) c)^* \iota_{|c^\top| - k}) \parallel (\mu \lambda b. b \ll k)^* ((\pi \lambda b. b \upharpoonright k) c)^{\circ - 1} \in \mathcal{P}(\mathbb{N}^*)^*$$

still for a fixed choice of k . To obtain a set of decompositions over the range of $k \in \mathcal{R}(\iota_{|c^\top| - 2}^1)$, let a function $v_8 : \mathcal{P}(\mathbb{N}^*) \rightarrow (\mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*))$ defined by

$$v_8 = \lambda c. \mu \lambda k. ((\lambda i. (\mu \lambda b. (b \upharpoonright k) \parallel \langle b_{k+i} \rangle) c)^* \iota_{|c^\top| - k}) \parallel (\mu \lambda b. b \ll k)^* ((\pi \lambda b. b \upharpoonright k) c)^{\circ - 1} \quad (11.29)$$

induce the result $(v_8 c) \mathcal{R}(\iota_{|c^\top| - 2}^1) \in \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*)$ in the definition

$$\nabla_{\dot{c}} = \lambda c. (\lambda i. \langle \emptyset, (v_8 c) \mathcal{R}(\iota_{|c^\top| - 2}^1) \rangle_i) \delta_{\epsilon}^{\mathbb{U}_s c} \delta_{\epsilon}^{\iota_3^{\uparrow \{c^\top\}}} - (\mu \mathbb{U}_s) \mathcal{P}(\mathcal{P}(\mathbb{N}^*))$$

mentioned in Equation 11.28, which satisfies $\nabla_{\dot{c}} c = \emptyset$ for planar or separable specifications c , and excludes separable results for similar reasons to Equation 11.27.

Inverse decomposition functions

To recover the coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ from a crossbar decomposition $d = \mathbb{U}_{\dot{c}} c \in \mathcal{P}(\mathbb{N}^*)^*$, we note from Equation 11.29 that each point $b \in d_i$ of a back end block specification d_i is derived originally from a member of c with a prefix of length k deleted from it, for a value of k that is easy to deduce from d , so restoring the right prefix to all of them should suffice. The set of prefixes is not much harder to obtain by inspecting the first k coordinates of all front end block points, and the lexicographically i -th prefix corresponds to the i -th back end block d_{n+i} , where n is the number of front end blocks, also easy to deduce from d .

Less vaguely, we have $k = |(d_0)^\top| - 1$, the predecessor of the front end block dimensionality as discussed previously, and a number of front end blocks

$$n = |(d_{|d|-1})^\top|$$

equal to the back end block dimensionality, leading to the list $a = v_9 d \in (\mathbb{N}^k)^{|d| - n}$ of prefixes ordered lexicographically for $v_9 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathbb{N}^{**}$ given by

$$v_9 = \lambda d. ((\mu \lambda b. b \upharpoonright |(d_0)^\top| - 1) \cup \mathcal{R}(d \upharpoonright |(d_{|d|-1})^\top|))^{\circ - 1} \quad (11.30)$$

and hence a prefix a_i for each back end block specification d_{n+i} for i ranging from 0 through $|a| - 1$. The original specification $c = \ddot{\mathbb{U}}_{\dot{c}}^{-1} d$ is then recoverable by

$$\ddot{\mathbb{U}}_{\dot{c}}^{-1} = \lambda d. (\lambda a. (\lambda n. \bigcup_{i \in \mathcal{D}(a)} a_i \parallel d_{n+i}) |d| - |a|) v_9 d \quad (11.31)$$

where the concatenation $a_i \parallel d_{n+i} \in \mathcal{P}(\mathbb{N}^{|c^\top|})$ of a list $a_i \in \mathbb{N}^k$ with a set of lists $d_{n+i} \in \mathcal{P}(\mathbb{N}^{|c^\top|-k})$ may be interpreted according to [Equation 7.3](#).

However, this inverse decomposition function is not completely satisfactory because it ignores all but the first k coordinates of each point in any front end block. If the rest of the coordinates are not consistent with a crossbar decomposition, then the coordinates c obtained by [Equation 11.31](#) should not be inferred. The exact requirement for the j -th front end block is that each point with a prefix of a_i should have a member of the set of inputs along the j -th axis of the i -th back end block

$$((d_{n+i})^\top)_j \in \mathcal{P}(\mathbb{N})$$

as its next and last coordinate prior to the truncation to length k inherent in [Equation 11.30](#), so that the whole j -th front end block specification should be

$$\bigcup_{i \in \mathcal{D}(a)} a_i \parallel ((d_{n+i})^\top)_j^1 \in \mathcal{P}(\mathbb{N}^*)$$

and the whole front end should be

$$f = (\lambda j. \bigcup_{i \in \mathcal{D}(a)} a_i \parallel ((d_{n+i})^\top)_j^1)^* \iota_{|n|} \in \mathcal{P}(\mathbb{N}^*)^*.$$

A better inverse decomposition function $\dot{\mathcal{U}}_c^{-1} : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^*)$ satisfying $\dot{\mathcal{U}}_c^{-1} d = \emptyset$ when this condition fails but devolving to the previous one otherwise can be defined as follows.

$$\dot{\mathcal{U}}_c^{-1} = \lambda d. (\lambda a. (\lambda n. (\lambda f. \langle \emptyset, \ddot{\mathcal{U}}_c^{-1} d \rangle_{\delta_f^{d+n}}) (\lambda j. \bigcup_{i \in \mathcal{D}(a)} a_i \parallel ((d_{n+i})^\top)_j^1)^* \iota_{|n|} |d| - |a|) v_9 d \quad (11.32)$$

A further improvement on this inverse decomposition function would refrain from invalid inferences due to separable decompositions d and exclude anomalous or undefined results if d is too long or too short to be a valid crossbar decomposition. Both [Equation 11.31](#) and [Equation 11.32](#) rely on the assumption of

$$n + |a| = |(d_{|d|-1})^\top| + |v_9 d| = |d|$$

and therefore $|d| > 0$ for the result to be well defined. A function satisfying $\mathcal{U}_c^{-1} d = \emptyset$ even without these conditions met but otherwise matching the previous one would be best to define as follows.

$$\mathcal{U}_c^{-1} = \lambda d. (\lambda e. \langle \emptyset, \langle \lambda v. \langle \emptyset, \dot{\mathcal{U}}_c^{-1} d \rangle_{\delta_v^{|d|}} \rangle_{\delta_e^d} |d| \rangle_{\delta_e^d} |d| \rangle_{\delta_e^d} |d| + |v_9 d|, \emptyset \rangle_{\delta_e^d} \rangle_{\delta_e^d} \mathcal{U}_s^{-1} d$$

Combining form

The crossbar combining form $\Omega_c : \mathcal{P}(\mathbb{N}^*) \times \mathbb{H}^* \times \mathbb{H}^* \rightarrow \mathbb{H}$ takes a triple $(d, x, y) \in \mathcal{P}(\mathbb{N}^*) \times \mathbb{H}^* \times \mathbb{H}^*$ consisting of a crossbar decomposition d and two lists of sparse decision waits x and y to a sparse decision wait $\Omega_c(d, x, y) \in \mathbb{H}$ with coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ provided that

$$d = \mathcal{U}_c c \in \mathcal{P}(\mathbb{N}^*)^*$$

is derived from c by [Equation 11.28](#), each x_i has coordinates $(\eta d)_i$ for $0 \leq i < |x|$, and each y_i has coordinates $(\eta d)_{|x|+i}$ for $0 \leq i < |y|$, where $|x| + |y|$ is equal to $|d|$.

One way to specify the crossbar combining form with respect to formal parameters d , x , and y as above is by a parallel combination of a FORK network and a bus connected through a front permutation network to the combination of front end blocks $(\mathcal{F} \mathbf{R}) x$, connected further through a central permutation network to the combination of back end blocks $(\mathcal{F} \mathbf{R}) y$ as shown in the following expression.

$$\Omega_c(d, x, y) = \mathbf{R}(E_1 E_0 d, |^{b E_3 d}|) \xrightarrow{(\mathring{b} b^* \langle E_2 E_0 d, E_3 d \rangle^T)^{-1}} (\mathcal{F} \mathbf{R}) x \xrightarrow{E_4 d} (\mathcal{F} \mathbf{R}) y \quad (11.33)$$

The rest of this section examines each of aspect of this construction at greater length.

FORK network To focus momentarily on the input arities of the front end blocks x , we consider only the front end block specifications $d \vdash n$, where $n = |x| = |(d_{|d|-1})^T|$ is the number of front end blocks as noted previously, and even more specifically on only the first k dimensional inputs to each front end block, where $k = |(d_0)^T| - 1$ is the parameter determining much of the rest of the decomposition as also discussed previously. By truncating the last coordinate from each point $b \in d_j$, we arrive at this goal via the expression

$$(\lambda t. t^T)^* (\mu \lambda b. b \vdash |b| - 1)^* d \vdash |(d_{|d|-1})^T| \in \mathcal{P}(\mathbb{N})^{**}$$

referring to a list of lists of sets of globally scoped input terminal numbers for the front end blocks x , with one term for each of n blocks and one set of input terminal numbers within each term for each of k dimensions. A list indexed by dimensions first and then block numbers is somewhat more convenient and easily obtained as the transpose $E_0 d$ of the expression above for a function

$$E_0 : \mathcal{P}(\mathbb{N})^{**} \rightarrow \mathcal{P}(\mathbb{N})^{**}$$

defined by

$$E_0 = \lambda d. ((\lambda t. t^T)^* (\mu \lambda b. b \vdash |b| - 1)^* d \vdash |(d_{|d|-1})^T|)^T. \quad (11.34)$$

The externally visible inputs to the crossbar combination begin with the first input in the first dimension, followed immediately by subsequent inputs in the first dimension, followed by all inputs in the next dimension, and so on, with each of the first k dimensional inputs connected to a FORK tree as shown in Figure 10.17. However, in a sparse decision wait, the FORK output arities may vary. In particular, the arity of the l -th FORK tree in the i -th dimension is equal to the number of terms in $(E_0 d)_i$ having l as a member, this being the number of front end blocks possessing an input terminal numbered l along their i -th axis. This arity could be expressed succinctly as the cardinality of the projection

$$|(\mathring{b}(E_0 d)_i) \uparrow (\mathcal{P}(\mathbb{N}) - \mathcal{P}(\mathbb{N} - \{l\}))|$$

for this particular combination of dimension i and input l relative to i . Letting $t = \mathring{b}(E_0 d)_i \in \mathbb{N}^*$ denote the list of all input terminal numbers in the i -th dimension (which necessarily contains duplicates if the same terminal number l appears on multiple blocks), we can scale up production by making the list

$$(\lambda l. \text{FORK } |t \uparrow (\mathcal{P}(\mathbb{N}) - \mathcal{P}(\mathbb{N} - \{l\}))|)^* \mathcal{R}(t)^{\circ-1} \in \mathbb{H}^*$$

of all FORK trees driven by the i -th dimensional input bus in the right order with the right arity for each. To make that a list of lists covering all k dimensions $0 \leq i < k$, we have

$$(\lambda t. (\lambda l. \text{FORK } |t \uparrow (\mathcal{P}(\mathbb{N}) - \mathcal{P}(\mathbb{N} - \{l\}))|)^* \mathcal{R}(t)^{\circ-1})^* \mathring{b}^* E_0 d \in \mathbb{H}^{**}$$

from which the whole FORK network follows as the parallel combination $E_1 E_0 d$ for a function

$$E_1 : \mathcal{P}(\mathbb{N})^{**} \rightarrow \mathbb{H}$$

given by

$$E_1 = (\mathcal{F} \mathbf{R}) \circ b \circ (\lambda t. (\lambda l. \text{FORK } |t \uparrow (\mathcal{P}(\mathbb{N}) - \mathcal{P}(\mathbb{N} - \{l\}))|)^* \mathcal{R}(t)^{\circ-1})^* \circ b^*$$

which makes modest progress towards [Equation 11.33](#).

Front permutation network Although the number of outputs from the FORK network derived above matches the number of inputs to the first k dimensions of the front end blocks $(\mathcal{F} \mathbf{R}) x$, the outputs are not in the right order to be connected directly without a permutation network. The front end requires all inputs to the first block x_0 first, followed by the inputs to the next block, *etc.*, and the inputs local to each block are ordered by their dimensions, including not just the first k but also the remaining dimension. On the other hand, the outputs from the FORK network are ordered mainly by dimensions, and then by the individual FORK tree within each dimension, each with various outputs destined for possibly non-consecutive blocks, and only for the first k dimensions.

Constructing the permutation is manageable in two steps, with the first step focusing on a typical l -th input in the i -th of the first k dimensions to the j -th block x_j , for which we seek the output terminal on the FORK network driving it. Letting $e = E_0 d$ by [Equation 11.34](#) denote the list of lists of sets of input terminal numbers on all blocks x grouped by dimensions and imagining the FORK outputs to be numbered globally and consecutively in the order described above, we note first that the FORK terminal number driving our hypothetical input must be at least

$$|\overset{\circ}{b} b(e \uparrow i)|$$

the number of all FORK outputs in dimensions preceding the i -th. Furthermore, it must exceed this number by at least

$$|(\overset{\circ}{b} e_i) \uparrow \mathcal{R}(l_l)|$$

the total number of outputs from all FORK trees preceding the l -th FORK tree in the i -th dimension, which we infer indirectly by counting the number of i -th dimensional inputs numbered less than l on any block. Finally, it must be further offset by

$$|(\overset{\circ}{b}(e_i \uparrow j)) \uparrow \{l\}|$$

the number of outputs from the l -th FORK tree in the i -th dimension connected to blocks preceding the j -th, or in other words the number of i -th dimensional inputs numbered l on blocks numbered less than j . The set of all FORK output terminal numbers associated with any input terminal $l \in e_{ij}$ along the i -th axis of the j -th block x_j is therefore

$$(\mu \lambda l. |\overset{\circ}{b} b(e \uparrow i)| + |(\overset{\circ}{b} e_i) \uparrow \mathcal{R}(l_l)| + |(\overset{\circ}{b}(e_i \uparrow j)) \uparrow \{l\}|) e_{ij} \in \mathcal{P}(\mathbb{N})$$

and a list of these sets indexed by the corresponding dimension i for $0 \leq i < k$ is

$$(\lambda i. (\mu \lambda l. |\overset{\circ}{b} b(e \uparrow i)| + |(\overset{\circ}{b} e_i) \uparrow \mathcal{R}(l_l)| + |(\overset{\circ}{b}(e_i \uparrow j)) \uparrow \{l\}|) e_{ij})^* \iota_{|e|} \in \mathcal{P}(\mathbb{N})^*$$

for a fixed block x_j . The sublist of the desired permutation due to this block would be the ordered list of all FORK terminal numbers in these sets

$$\overset{\circ}{b}(\lambda i. (\mu \lambda l. |\overset{\circ}{b} b(e \mid i)| + |(\overset{\circ}{b} e_i) \uparrow \mathcal{R}(l_i)| + |(\overset{\circ}{b}(e_i \mid j)) \uparrow \{l\}|) e_{ij})^* \iota_{|e|} \in \mathbb{N}^*$$

designated hereafter as $(E_2 E_0 d)_j \in \mathbb{N}^*$ for a function $E_2 : \mathbb{N}^{**} \rightarrow \mathbb{N}^{**}$ defined by

$$E_2 = \lambda e. (\lambda j. \overset{\circ}{b}(\lambda i. (\mu \lambda l. |\overset{\circ}{b} b(e \mid i)| + |(\overset{\circ}{b} e_i) \uparrow \mathcal{R}(l_i)| + |(\overset{\circ}{b}(e_i \mid j)) \uparrow \{l\}|) e_{ij})^* \iota_{|e|})^* \iota_{|e\tau|}$$

which concludes the first step toward deriving the front permutation network.

The second step concerns the inputs along the last axis of each front end block. These input terminals are exposed externally and therefore not connected to the FORK network, but because they correspond to dimensions k through $|c^\top| - 1$ as seen from the outside, it is right that they be driven by a bus in parallel with the FORK network as in Equation 11.33. Hence we envision the output terminals from this bus as a continuation of the sequence of FORK output terminals, with the bus output terminals numbered starting from

$$\sum b(s^\top \mid k)$$

the total number of outputs from the FORK network. This sum is inferred from the total number of inputs in the first k dimensions of the front end blocks x , whose dimensions

$$s = \sigma^* \eta d \mid n \in \mathbb{N}^{**}$$

are based on $n = |(d_{|d|-1})^\top|$ as noted previously.

In these terms, the j -th block x_j has s_{jk} inputs along its last axis, which must be driven by s_{jk} bus lines starting from the output terminal numbered

$$(\sum b(s^\top \mid k)) + \sum (s_k^\top \mid j)$$

on the FORK network and bus combination, the latter sum being due to bus lines connected to lower numbered blocks. These connections account for a sublist

$$\iota_{s_{jk}}^{(\sum b(s^\top \mid k)) + \sum (s_k^\top \mid j)}$$

of the desired permutation, designated more briefly as $(E_3 d)_j \in \mathbb{N}^*$ for $E_3 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathbb{N}^{**}$ with

$$E_3 = \lambda d. (\lambda(k, n). (\lambda s. (\lambda j. \iota_{s_{jk}}^{(\sum b(s^\top \mid k)) + \sum (s_k^\top \mid j)})^* \iota_n) (\sigma^* \eta d \mid n)) (|(d_0)^\top| - 1, |(d_{|d|-1})^\top|)$$

which also makes the bus expressible as $|\overset{|b}{E_3} d|$, its width equal to the combined widths of all buses connected to the last axes of front end blocks in x .

To assemble the whole permutation, we have to concatenate each $(E_2 E_0 d)_j$ with the corresponding $(E_3 d)_j$ to get the whole sublist for the j -th block x_j for all $0 \leq j < n$ in

$$b^*(\langle E_2 E_0 d, E_3 d \rangle^\top) \in \mathbb{N}^{**}$$

and then flatten it again and invert it to get a permutation

$$(b b^*(\langle E_2 E_0 d, E_3 d \rangle^\top))^{-1} \in \mathbb{N}^*$$

that assigns an input terminal number on $(\mathcal{F} \mathbf{R}) x$ to each output terminal number on the FORK network and bus as required in Equation 11.33.

Central permutation network The last needed feature of the crossbar combination is the permutation network that connects the front end blocks x to the back end blocks y . For this part, let the list $s_i \in \mathbb{N}^*$ denote the dimensions of the i -th back end block y_i with

$$s = \sigma^* \eta d \ll n \in \mathbb{N}^{**}$$

and $|s| = |y| = |d| - n$ for $n = |x| = |(d_{|d|-1})^\top|$ as usual. Then each front end block x_j for $0 \leq j < n$ can be viewed as driving

$$k = |(d_0)^\top| - 1 = |s|$$

output buses such that the i -th bus has s_{ij} lines.

We would like to connect this bus to the j -th dimensional axis of the i -th back end block y_i . In the sequence of input terminals to the combination $(\mathcal{F} \mathbf{R}) y$ used in Equation 11.33, those of y_i start at the position numbered $\sum_{b(s \uparrow i)}$, the total number of input terminals on back end blocks numbered less than i , and those of its j -th axis start at

$$a = \sum(s_i \uparrow j) \parallel b(s \uparrow i)$$

with the dimensions $s \in \mathbb{N}^{**}$ as denoted above. A sequence of s_{ij} consecutive numbers starting from a in these terms specifies the destination terminals of i -th bus from x_j as a sublist of the permutation describing the central permutation network.

For the sublists of the permutation to be ordered consistently with the buses from $(\mathcal{F} \mathbf{R}) x$ whose destinations they specify, they should start with that of the initial bus from x_0 followed by the rest of the sublists due to x_0 , and those of other blocks x_j should similarly be bundled together in a list

$$b(\lambda i. (\lambda a. \iota_{s_{ij}}^a) \sum(s_i \uparrow j) \parallel b(s \uparrow i))^* \iota_{|s|} \in \mathbb{N}^*$$

for each $0 \leq j < n$ concatenated into a list

$$b(\lambda j. b(\lambda i. (\lambda a. \iota_{s_{ij}}^a) \sum(s_i \uparrow j) \parallel b(s \uparrow i))^* \iota_{|s|})^* \iota_n \in \mathbb{N}^*$$

accounting for all front end blocks x_j . Hence we define the whole central permutation $E_4 d \in \mathbb{N}^*$ appearing in Equation 11.33 in terms of a function $E_4 : \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathbb{N}^*$ given by

$$E_4 = \lambda d. (\lambda n. (\lambda s. b(\lambda j. b(\lambda i. (\lambda a. \iota_{s_{ij}}^a) \sum(s_i \uparrow j) \parallel b(s \uparrow i))^* \iota_{|s|})^* \iota_n) (\sigma^* \eta d \ll n)) |(d_{|d|-1})^\top|$$

which also concludes a discussion of multidimensional sparse decision waits overall to the extent of providing a workable *ad hoc* methodology for their synthesis, but may leave something to be desired by readers who prefer a more comprehensive approach and are not overly averse to further reading.

11.6 Optimization

The loose end throughout this chapter has been the incompletely specified functions \mathcal{U}_v through \mathcal{U}_c needed to narrow down the range of possible implementations of MSDW c to a single one. Suggested heuristics such as choosing \mathcal{U}_c to maximize the area of the empty quadrant in Figure 11.6 are not guaranteed optimal, and even if they were locally optimal they might not always procure global optimality in the sense discussed on page 298. A decomposition function that unflinchingly maps any coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ to the optimal sparse decision wait having those coordinates, or at least the optimal one with respect to some fixed metric, needs to be of a different type than those previously considered. The rest of this section is about constructing and using one.

11.6.1 Sparse global decompositions

The distinguishing feature of such a function is that it indicates all of the decompositions in a complete nested hierarchy of building blocks with commensurate discretion. For example, instead of always maximizing the empty quadrant in an enmeshed decomposition, it could do so in some cases and not others depending on whatever best suits the building blocks below. More crucially, the decompositions of the blocks below can depend in effect not just on their own coordinates but on their roles in the hierarchy. This agenda may benefit from the following more explicit formulation.

Hierarchical structures The expressiveness envisioned above presupposes a hierarchical structure of a type convenient to model as a member of the set of ordered trees

$$\dot{\mathcal{S}} = \mathcal{O}(\mathbb{N}^{**} \times (\mathbb{N}^{**} \cup \mathcal{P}(\mathbb{N}^*)^*))$$

containing all **sparse global decompositions**, which also happens to be a superset of the set \mathcal{S} of dense global decompositions defined in [Section 10.6.1](#). Any sparse global decomposition

$$t = ((p, d), s) \in \dot{\mathcal{S}}$$

consists of a list of permutations $p \in \mathbb{N}^{**}$, a dense or sparse local decomposition

$$d \in \mathbb{N}^{**} \cup \mathcal{P}(\mathbb{N}^*)^*$$

and a list $s \in \dot{\mathcal{S}}^*$ of sparse global decompositions with the intended significance that $d \in \mathbb{N}^{**}$ means t describes a dense decision wait as do all subtrees listed in s , and p determines an optional transformation by [Equation 10.25](#) or [Equation 11.5](#).

Well formed decompositions We can characterize valid sparse global decompositions precisely as those for which non-empty dimensions or coordinates can be inferred by way of a function

$$\dot{\psi} : \mathcal{P}(((\mathcal{P}(\mathbb{N}^*)^* \times \mathcal{P}(\mathbb{N}^*)^*) \rightarrow \mathcal{P}(\mathbb{N}^*))) \rightarrow (\dot{\mathcal{S}} \rightarrow (\mathbb{N}^* \cup \mathcal{P}(\mathbb{N}^*)))$$

analogous to the function ψ defined by [Equation 10.26](#) for dense decision waits as follows.

$$\dot{\psi} = \lambda h. \Lambda \lambda((p, d), v). \begin{cases} (\lambda i. \langle \epsilon, (\lambda j. \langle \epsilon, (\lambda r. |p_{r+1}|)^* p_{0j}) \rangle_i \delta_{|p_0|}^{p+1}) \Delta_g(p, d, v) & \text{if } \{d, v\} \subset \mathbb{N}^{**} \\ (\hat{\psi} p) \bigcup_{f \in h} f(d, (\lambda t. (\lambda i. \langle t, \mathcal{R}(\bar{t}_t) \rangle_i) \delta_{\mathbb{N}^*}^{\mathbb{N}^* \cup \{t\}})^* v) & \text{if } d \in \mathcal{P}(\mathbb{N}^*) \\ \emptyset & \text{otherwise} \end{cases}$$

That is, if d is a dense decomposition and the values inferred for all subtrees in v are lists of dimensions, then we infer a list of dense decision wait dimensions from the permutation lengths subject to Δ_g by [Equation 10.35](#), but if d is a sparse decomposition, then where applicable we promote any dimensions $t \in \mathbb{N}^*$ inferred from the subtrees v to coordinates $\mathcal{R}(\bar{t}_t) \in \mathcal{P}(\mathbb{N}^*)$ and infer the coordinates overall by members $f \in h$ of the set of functions $h \in \mathcal{P}(((\mathcal{P}(\mathbb{N}^*)^* \times \mathcal{P}(\mathbb{N}^*)^*) \rightarrow \mathcal{P}(\mathbb{N}^*)))$ parameterizing

$$\dot{\psi} h : \dot{\mathcal{S}} \rightarrow \mathbb{N}^* \cup \mathcal{P}(\mathbb{N}^*).$$

The additional term $\hat{\psi} p$ ([Equation 11.7](#)) transforms the coordinates that would otherwise be inferred from d and v consistently with those of the sparse decision wait to be generated from the decomposition as described presently in [Section 11.6.2](#).

Characterizing valid sparse global decompositions now reduces to the question of suitable coordinate inference functions $f : \mathcal{P}(\mathbb{N}^*)^* \times \mathcal{P}(\mathbb{N}^*)^* \rightarrow \mathcal{P}(\mathbb{N}^*)$ whereby $f(d, v) \in \mathcal{P}(\mathbb{N}^*)$ is non-empty for valid combinations of d and v . (Incompatible permutations p would necessarily yield an empty result by [Equation 11.6](#) and [Equation 11.8](#).) There are three main possibilities.

- If the decomposition describes a degenerate sparse decision wait, there should be a single subtree describing a dense decision wait and a single decomposition term $d_0 \subseteq v_0$ matching the coordinates given by

$$\Delta_t(d, v) = (\lambda i. \langle \emptyset, (\lambda j. \langle \emptyset, d_0 \rangle_j) \delta_{v_0}^{(d_0)^{\text{tr}}} \rangle_i) \delta_1^{|d||v|}.$$

- If the decomposition at the top level describes anything other than an enmeshed combination, then the coordinates of the subtrees should coincide with the decomposition d subject to local renumbering, and the coordinates inferred for the result should be given by one of the inverse decomposition functions derived throughout this chapter.

$$\Delta_n(d, v) = (\lambda i. \langle \emptyset, (\lambda s. \bigcup_{f \in s} f d) \{ \mathcal{U}_s^{-1}, \mathcal{U}_v^{-1}, \mathcal{U}_h^{-1}, \mathcal{U}_d^{-1}, \mathcal{U}_c^{-1} \} \rangle_i) \delta_v^{\eta d}$$

- If the decomposition describes an enmeshed combination, then non-empty coordinates $\Delta_e(d, v)$ can be inferred according to the inverse enmeshed decomposition function only if d and v both meet this raft of conditions by [Equation 11.22](#).

$$\Delta_e(d, v) = (\lambda i. \langle \emptyset, (\lambda j. \langle \emptyset, \mathcal{U}_e^{-1} d \rangle_j) \delta_{v \ll 3}^{\eta d \ll 3} \delta_{\langle v_1, v_2 \rangle}^{\eta \dot{\mathcal{U}}_e(d \ll 3)} \delta_{v_0}^{\{0,1\}^2 - \{1\}^2} \rangle_i) \delta_6^{|v|}$$

With a minor abuse of notation, we can call a sparse global decomposition $t \in \dot{\mathbb{S}}$ valid or **well formed** if and only if it satisfies

$$|\dot{\psi}\{\Delta_t, \Delta_n, \Delta_e\} t| > 0$$

whether $\dot{\psi}\{\Delta_t, \Delta_n, \Delta_e\} t$ is a list of dimensions or a set of points, and regard this value as the coordinates or dimensions of the decision wait t describes.



11.6.2 General combining form

A combining form $\hat{\Omega}_{\dot{g}} : \dot{\mathbb{S}} \rightarrow \mathbb{H}$ confers a circuit semantics on sparse global decompositions $t \in \dot{\mathbb{S}}$ that are well formed based on the general dense combining form $\Omega_g : \mathbb{S} \rightarrow \mathbb{H}$ defined by [Equation 10.36](#) and the transformational operator φ defined by [Equation 11.5](#).

$$\hat{\Omega}_{\dot{g}} = \Lambda \lambda((p, d), v). \left\{ \begin{array}{ll} \varphi(p, d_0) \Omega_n(d_0, v_0) & \text{if } |d||v| = 1 \\ \varphi(p, \mathcal{U}_s^{-1} d) \Omega_s(d, v) & \text{if } \mathcal{U}_s^{-1} d \neq \emptyset \\ \varphi(p, \mathcal{U}_v^{-1} d) \Omega_v(d, v_0, v_1) & \text{if } \mathcal{U}_v^{-1} d \neq \emptyset \\ \varphi(p, \mathcal{U}_h^{-1} d) \Omega_h(d, v_0, v_1) & \text{if } \mathcal{U}_h^{-1} d \neq \emptyset \\ \varphi(p, \mathcal{U}_e^{-1} d) \Omega_e(d, v_0, \langle v_1, v_2 \rangle, v \ll 3) & \text{if } \mathcal{U}_e^{-1} d \neq \emptyset \\ \varphi(p, \mathcal{U}_d^{-1} d) \Omega_d(d, v \uparrow |v| - 1, v_{|v|-1}) & \text{if } \mathcal{U}_d^{-1} d \neq \emptyset \\ \varphi(p, \mathcal{U}_c^{-1} d) (\lambda n. \Omega_c(d, v \uparrow n, v \ll n)) |(d_{|d|-1})^{\text{tr}}| & \text{if } \mathcal{U}_c^{-1} d \neq \emptyset \\ \Omega_g((p, d), v) & \text{otherwise} \end{array} \right. \quad (11.35)$$

As noted above, degenerate sparse decision waits are expressible by trees $((p, d), s) \in \dot{\mathbb{S}}$ having exactly one subtree $s_0 \in \mathbb{S}$ such that $v_0 = \hat{\Omega}_{\dot{g}} s_0 = \hat{\Omega}_g s_0 \in \mathbb{H}$ is dense with dimensions σd_0 .

11.6.3 Decomposition strategies

To revisit the issue motivating this discussion in more precise terms, a decomposition function capable of specifying globally optimized sparse decision waits as a rule would be one like

$$\mathcal{U}_g : \mathcal{P}(\mathbb{N}^*) \rightarrow \dot{\mathcal{S}}$$

taking coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ to a well formed sparse global decomposition $t = \mathcal{U}_g c \in \dot{\mathcal{S}}$ such that

$$c = \psi \{ \Delta_t, \Delta_n, \Delta_e \} \mathcal{U}_g c$$

holds for any coordinates c with $|\sigma c| > 0$, and would induce the more capable multidimensional sparse decision wait generating function

$$\text{MSDW} = \hat{\Omega}_g \circ \mathcal{U}_g \quad (11.36)$$

by [Equation 11.35](#) in place of the one considered in [Section 11.5](#). A function like \mathcal{U}_g meeting these conditions may be called a **sparse global decomposition strategy** hereafter. Now all we need is to find one, which is done in this section by enumerating the possibilities first and then narrowing them down.

Enumeration

Ultimately an informed choice of a strategy depends on searching or sampling the space of well formed sparse global decompositions in some way. Whereas enumerating dense decompositions is a routine exercise ([item 4](#), page [313](#)), doing the same for sparse decompositions is a bit more of a challenge. One way to approach it is by taking each component p , d , and s of a sparse global decomposition $((p, d), s)$ in turn.

Local decomposition An easier task than generating all possible trees $((p, d), s) \in \dot{\mathcal{S}}$ would be to generate just the set $G_0 c \in \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*)$ of all possible sparse local decompositions $d \in \mathcal{P}(\mathbb{N}^*)^*$ for coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ by a function $G_0 : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}^*)^*)$ defined as

$$G_0 = \lambda c. \{ \langle c \rangle \} \cup (\lambda g. \bigcup_{f \in g} f c) \{ \nabla_s, \nabla_v, \nabla_h, \nabla_e, \nabla_d, \nabla_i \}$$

in terms of functions ∇_s through ∇_i derived throughout this chapter. The obligatory singleton decomposition $d = \langle c \rangle$ pertains to the degenerate case.

Permutations If a member $d \in G_0 c$ were to appear in a sparse global decomposition $((p, d), s) \in \dot{\mathcal{S}}$ for a decision wait with coordinates c , what permutations p would be suitable accompaniments? Anything other than a list of identity permutations would usually invalidate the result by changing its coordinates to something other than c . However, if instead of $G_0 c$ we started with decompositions $G_0 (\hat{\varphi} p') c$ for some arbitrary list of permutations p' , then any permutations p that effect an inverse transformation to p' would leave the result intact. Technically p' can not be completely arbitrary but must be a member of $G_1 c$ with $G_1 : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^{**})$ given by

$$G_1 = \lambda c. \left((\lambda n. \mathbb{N}^n \cap (\mathcal{D}(t_n) \mapsto \mathcal{R}(t_n)))^* (|\sigma c| : \sigma c) \right)^\top$$

where $\mathbb{N}^n \cap (\mathcal{D}(t_n) \rightsquigarrow \mathcal{R}(t_n))$ refers to the set of all permutations of length n expressed as injective lists, or else p' would not match the dimensions of c . In any case, the list p of permutations appearing in the root of the sparse global decomposition $((p, d), s)$ could then be any member of $G_2(p', c)$ for the function $G_2 : \mathbb{N}^{**} \times \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^{**})$ given by the deceptively simple definition

$$G_2 = \lambda(p', c). \{p \in \mathbb{N}^{**} \mid c = (\hat{\phi} p) (\hat{\phi} p') c\}$$

for which the author is unable to propose any reasonable algorithm. Fortunately it always includes and rarely exceeds the approximation obtained by individually inverting the permutations in p' and neglecting possible synergies peculiar to c .²

$$G_2 = \lambda(p', c). \{(p'_0)^{-1} : ((\lambda q. q^{-1})^* ((p' \ll 1) \circ p'_0))\}$$

In summary, the original lone coordinate specification $c \in \mathcal{P}(\mathbb{N}^*)$ now gives rise to a whole set of sparse global decomposition roots $G_3 c$ according to $G_3 : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N}^*)^*)$ expressed less awkwardly with the primes dropped.

$$G_3 = \lambda c. \bigcup_{p \in G_1 c} G_2(p, c) \times G_0(\hat{\phi} p) c.$$

Subtrees Having enumerated the roots (p, d) of all sparse global decompositions $((p, d), s) \in \dot{\mathbb{S}}$ given coordinates c , we have only the subtrees s left to assign. If d were always a unit list then it would be easy, with s invariably a member of

$$\{t \in \mathbb{S} \mid (\psi \Delta_g) t = \sigma d_0\}^1$$

the set of unit lists of dense decompositions with dimensions σd_0 befitting a degenerate sparse decision wait with coordinates d_0 by Equation 10.26 and Equation 10.35. More generally we might have to assume a recursively defined function

$$\nabla_{\dot{g}} : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathcal{P}(\dot{\mathbb{S}})$$

taking coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ to the set of all sparse global decompositions $\nabla_{\dot{g}} c \in \mathcal{P}(\dot{\mathbb{S}})$ describing decision waits with coordinates c . Then for $|d| > 1$, the list s of subtrees would be a member of

$$(\nabla_{\dot{g}}^* \eta d)^{\mathsf{T}}$$

the set of all lists of sparse global decompositions wherein the i -th term describes a decision wait with coordinates $(\eta d)_i$, the coordinates of the locally renumbered i -th term of the value d in the root, for all $0 \leq i < |d|$. The recurrence overall therefore should look something like this.

$$\nabla_{\dot{g}} = \lambda c. \bigcup_{(p, d) \in G_3 c} \{ \{(p, d)\} \times \langle (\nabla_{\dot{g}}^* \eta d)^{\mathsf{T}}, \{t \in \mathbb{S} \mid (\psi \Delta_g) t = \sigma d_0\}^1 \rangle_{\delta_1^{|d|}} \}$$

²Whoever is intrigued should chase up Pólya's enumeration theorem [69, 287] or see [103] for an accessible overview of related topics in combinatorics and group theory.

Selection

Do these efforts lead to a straight answer to the question of what sparse global decomposition strategy $\mathcal{U}_{\dot{g}}$ should be plugged into [Equation 11.36](#) to yield optimum results? The element of discretion is now confined in principle to the matter of a cost metric $\|x\| \in \mathbb{R}$ considered desirable to minimize for circuits $x \in \mathbb{H}$. Subject to this modest requirement, the exact optimum sparse global decomposition strategy of

$$\mathcal{U}_{\dot{g}} = \lambda c. (\lambda(m, t). t) \min (\mu \lambda t. (\|\hat{\Omega}_{\dot{g}} t\|, t)) \nabla_{\dot{g}} c$$

is attainable with sufficient resources, but approximations might be unavoidable in practice.

11.7 Verification

Perhaps overdue is a discussion of what it would mean for the decision wait generating functions defined in this chapter and [Chapter 10](#) to be correct, and furthermore, whether they are. A clear answer to the first question is a matter of routine construction in terms of the theory developed in Part II. The second is beyond the scope of this book because its answer would require a hand written or at best a machine assisted proof. However, the correctness of an individual decision wait with respect to its specification can always be established automatically regardless of its provenance, and of course the same applies to any finite number of decision waits.



Correctness of a decision wait with respect to its specification amounts to the satisfaction of [Equation 8.34](#), the generalized refinement relation, with the specification on the left and the implementation on the right. The most transparent and least error prone way to describe the specification is as a Petri net modeled DI process expressed by the process combinators introduced in [Chapter 5](#). On the other side, the implementation is what we want to verify, so it might be constructed by hand, by a decision wait generating function, or by a wild guess, subject only to being either a netlist or a hierarchical block in $\mathbb{L} \cup \mathbb{H}$. The rest of this section focuses mostly on the process specification and returns to the questions of correctness at the last hurdle.

11.7.1 Alphabet ordering

A process behaving like a sparse decision wait with coordinates $c \in \mathcal{P}(\mathbb{N}^*)$ has to have one input or output symbol for each terminal on the decision wait, so it needs $\sum \sigma c$ input symbols and $|c|$ output symbols in its alphabet. Generalized refinement between a process and a circuit presupposes an alphabet ordering, so it might as well consist of the first $|c| + \sum \sigma c$ generic alphabet symbols by [Equation 8.27](#) in ascending order, denoted $\alpha c \in \mathbb{G}^*$ with $\alpha : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathbb{G}^*$ given by

$$\alpha = \lambda c. (\lambda a. \mathbb{G}^{\circ-1*} \iota_a) |c| + \sum \sigma c.$$

11.7.2 Input symbol assignment

Aligning the alphabet to the actual ordering of the terminals on the device is the next trick. With regard to the input alphabet, the terminals are organized into $|\sigma c|$ buses with the j -th bus having a width $(\sigma c)_j$ and the j -th of b_0 through $b_{|b|-1}$ coordinates in any point $b \in c$ referring to the b_j -th terminal on the j -th bus. The input alphabet symbol associated with this terminal must be distinct

from any of the $\sum((\sigma c) \upharpoonright j)$ symbols associated with a terminal on any of the j preceding buses and from the other $(\sigma c)_j - 1$ symbols associated with other terminals on the same bus. If αc is to list the input symbol associated with each terminal in ascending order of terminals on each bus in ascending order of buses, then it would be best to identify b_j with the alphabet symbol

$$(\alpha c) (b_j + \sum((\sigma c) \upharpoonright j))$$

in the forthcoming process combinator expression determined by c , or the whole point b with

$$(\lambda j. (\alpha c) (b_j + \sum((\sigma c) \upharpoonright j)))^* \iota_{|b|}.$$

11.7.3 Output symbol assignment

If we think of the output side as one large bus having a width $|c|$, then a point $b \in c$ as a whole refers to the n -th output bus line with $n = c^\circ b$ being the lexicographic ordinal of b with respect to c , which is natural to identify with the $(c^\circ b)$ -th output alphabet symbol

$$(\alpha c) ((c^\circ b) + \sum \sigma c).$$

A self-contained intermediate representation $p_0 c \in (\mathbb{G}^* \times \mathbb{G})^*$ of the specification c summarizes these assignments of input and output symbols with $p_0 : \mathcal{P}(\mathbb{N}^*) \rightarrow (\mathbb{G}^* \times \mathbb{G})^*$ defined by

$$p_0 = \lambda c. (\lambda b. ((\lambda j. (\alpha c) (b_j + \sum((\sigma c) \upharpoonright j)))^* \iota_{|b|}, (\alpha c) ((c^\circ b) + \sum \sigma c)))^* c^{\circ-1}$$

so that each term $(i, o) \in \mathcal{R}(p_0 c)$ has a list $i \in \mathbb{G}^*$ of input symbols on the left and the corresponding output symbol $o \in \mathbb{G}$ on the right.

11.7.4 Process specification

Each pair $(i, o) \in \mathbb{G}^* \times \mathbb{G}$ obtained above indicates that receiving the input signals in i and then transmitting the output signal o is one way for the decision wait to behave. Describing this behavior as a process, we have

$$\text{seq} ((\mathcal{F} \text{ par}) \text{ get}^* i, \text{ put } o) \in \mathbb{D}$$

and enumerating all such behaviors indicated in $p = p_0 c$ in the list of processes

$$(\lambda(i, o). \text{seq} ((\mathcal{F} \text{ par}) \text{ get}^* i, \text{ put } o))^* p \in \mathbb{D}^*$$

leads to a specification for a process that can repeatedly exhibit any of them by [Equation 3.5](#).

$$\text{loop} (\mathcal{F} \text{ alt}) (\lambda(i, o). \text{seq} ((\mathcal{F} \text{ par}) \text{ get}^* i, \text{ put } o))^* p \in \mathbb{D}$$

However, this specification as written describes a process that must arbitrate among contentious inputs, which is more than we demand of a decision wait. In practice a decision wait has the much easier job of interacting with a pleasant environment like

$$\text{loop} (\mathcal{F} \text{ alt}) (\lambda(i, o). \text{seq} ((\mathcal{F} \text{ par}) \text{ put}^* i, \text{ get } o))^* p)$$

which sends exactly one of the agreed input bursts i at a time and then politely waits for the acknowledgment o before sending another one. We make this assumption explicit by defining the

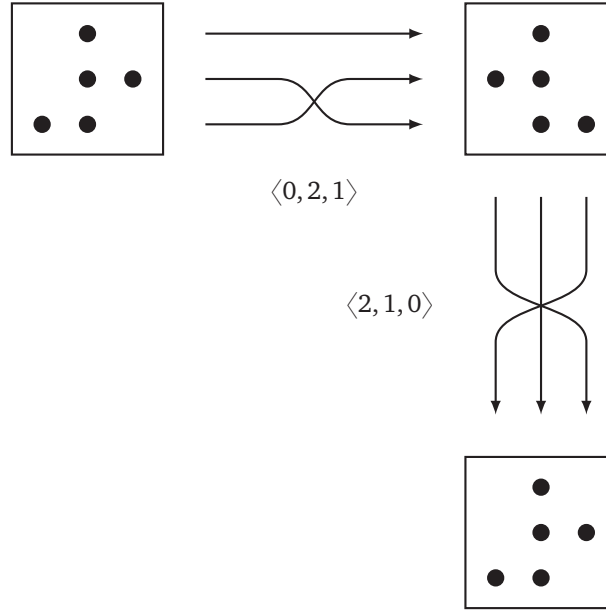


Figure 11.10: For this particular 3-by-3 sparse decision wait, permuting the rows by $\langle 0, 2, 1 \rangle$ and the columns by $\langle 2, 1, 0 \rangle$ yields a behaviorally equivalent result.

process specification as $p_1 p_0 c$ for $p_1 : (\mathbb{G}^* \times \mathbb{G})^* \rightarrow \mathbb{D}$ expressed using the `env` combinator (cf. Equation 9.16).

$$p_1 = \lambda p. \text{env} (\\ \text{loop } (\mathcal{F} \text{ alt}) (\lambda(i, o). \text{seq } ((\mathcal{F} \text{ par}) \text{ get}^* i, \text{put } o))^* p, \\ \text{loop } (\mathcal{F} \text{ alt}) (\lambda(i, o). \text{seq } ((\mathcal{F} \text{ par}) \text{ put}^* i, \text{get } o))^* p)$$

11.7.5 Correctness

Given that $p_1 p_0 c \in \mathbb{D}$ is an accurate description of the behavior expected of a purported sparse decision wait $\text{MSDW } c \in \mathbb{H}$ with coordinates $c \in \mathcal{P}(\mathbb{N}^*)$, the exact requirement for its correctness is

$$p_1 p_0 c \stackrel{\alpha c}{\sqsubseteq} \text{MSDW } c$$

by Equation 8.34. Whether the function MSDW yields correct results in other cases is irrelevant. However, if a sparse decision wait generating function $f : \mathcal{P}(\mathbb{N}^*) \rightarrow \mathbb{H}$ itself were on trial, then a formal statement of its correctness would be more like

$$\forall c \in \mathcal{P}(\mathbb{N}^*). |\sigma c| > 0 \Rightarrow p_1 p_0 c \stackrel{\alpha c}{\sqsubseteq} f c$$

which entails universal quantification over all valid coordinates c . As noted previously, the former is mechanically verifiable while the latter would require an *ad hoc* proof.

Idea management

1. If a separable decomposition corresponds to the connected components in a graph (page 324), in what similar way could an enmeshed decomposition be determined by a *clique*? (hint: The nodes of the graph are still rows of the decision wait, but the adjacency relation is different.)
2. Assuming union, intersection, subtraction, and comparison of sets of points are linear time operations, what would be the time complexity of a naively implemented algorithm for enmeshed decomposition that always maximizes the height of the empty quadrant in Figure 11.6?
3. The maximum clique problem is known to be NP-hard [30, 95, 260]. What stands in the way of fame and fortune for the first person to solve arbitrary instances efficiently by transforming them to enmeshed decomposition problems and solving them as proposed above?
4. With regard to the circuit $Z^2R(\text{PUSH}, \text{JOIN})$ used in the degenerate sparse decision wait construction (Figure 11.1), what would be the effects of
 - a) omitting the PUSH?
 - b) replacing the JOIN with a MERGE?
5. Further to the footnote on page 356, Figure 11.10 shows an example of a 3-by-3 sparse decision wait that can be transformed to itself by something other than a pair of identity permutations on its rows and columns.
 - a) How many 3-by-3 sparse decision waits are there?
 - b) How many are invariant with respect to some pair of row and column permutations not both the identity (*i.e.*, $\langle 0, 1, 2 \rangle$)?
 - c) How many are invariant with respect to more than one such pair?
 - d) Would allowing rotations affect anything?
 - e) Are analogous instances more common in higher dimensions, or less?
6. How good of an idea would it be to generalize enmeshed combination directly to higher dimensions as an alternative to the dendriform and crossbar combinations?
7. A manager distrusts the line of reasoning in Section 11.7 because the process specification $p_1 p_0 c$ is too complicated, but would feel a lot better about seeing a graphical representation of its Petri net model (especially if it were generated and optimized automatically). How should it look? How would an optimal transducer model look?



The general who wins the battle
makes many calculations in his
temple before the battle is fought.
The general who loses makes but
few calculations beforehand.

Sun Tzu

CHAPTER

12

ALL ABOUT ARBITERS

Arbiters are crucial to any application where multiple clients compete for limited resources, such as a network hub or a bus controller, and can have a major impact for better or worse on the overall quality of a design. One aspect of addressing this issue effectively is in the physical-level implementation of primitives such as the two-input ARB described in [Chapter 9](#), a matter of scrutiny from its earliest investigations [[179](#), [180](#), [245](#), [246](#), [310](#)] all the way through to contemporary research [[194](#), [204](#)]. The job of designing an arbiter does not end there, because the next problem is to combine these carefully engineered primitives into arbiters having the number of inputs required by the application (*i.e.*, the number of competing clients) in a way that optimizes whatever metrics are of interest. This chapter focuses exclusively on the latter aspect, taking the primitives as given.

Even with the focus thus narrowed, this topic may prove to be more than enough of a challenge all by itself. Readers in a hurry should probably stick with the multi-way arbiter design illustrated in [Figure 9.18](#) (called a *triangle mesh arbiter* in [[120](#)]) and hope for the best. Indeed, if there were ever a time to heed the counsel of textbook authors against undue preoccupation with asynchronous circuits [[175](#)], it would be now.

To expand less flippantly on this point, undoubtedly some readers are already painfully aware of the need for arbiters by being forced to use one as the only asynchronous part of an otherwise conventional synchronous design, and have probably helicoptered directly into this chapter to requisition some arbiter know-how and nothing else. This battle plan is questionable due to the reliance of this chapter on prior chapters. The best survival tactic for a reader in this position would be to skim the chapter long enough to be convinced, refer to [Section 9.4.2](#) to address the immediate need, and then ideally devise a new plan either to give the book a thorough reading in the distant future when time permits, or to retreat, retrench, and lower expectations to less ambitious circuit design projects from now on.



For those undeterred, the payoff of this chapter is a quantitative framework for optimizing the aforementioned metrics, which encompass fairness, contention, and critical path length, as well as any others the reader may be in a position to formulate. In other words, what is at stake is no less than a decision procedure for establishing whether one design surpasses another in a statistical sense with respect to any freely specified traffic patterns and performance criteria.

The culmination of this result in [Section 12.5](#) comes not without significant preparation in the intervening sections. Aside from a bit of admittedly non-standard but justifiably helpful notation proposed in [Section 12.1](#), this preparation consists mainly of playing the same game with arbiters as with decision waits in previous chapters, which can be summarized far more succinctly this time around thanks to its familiarity. That is, multi-way arbiters are described by hierarchical decompositions represented as members of a countable set \mathbb{A} of ordered trees ([Section 10.1](#)). A decomposition strategy $\mathcal{U}_a : \mathbb{N} \rightarrow \mathbb{A}$ that can be optimized relative to any fixed metric maps a given arity $n \in \mathbb{N}$ to the decomposition $\mathcal{U}_a n \in \mathbb{A}$ of an arbiter having that arity, a generalized combining form $\Omega_a : \mathbb{A} \rightarrow \mathbb{H}$ maps decompositions to the circuits they describe, and an arbiter generating function $\text{ARB} = \Omega_a \circ \mathcal{U}_a$ defined in terms of the preferred choice of \mathcal{U}_a maps an arity n to an optimized arbiter $\text{ARB } n \in \mathbb{H}$.

The middle parts of this chapter entail a proposed taxonomy of arbiter decompositions in [Section 12.2](#), a concept of time evolution formalized as a transfer function in [Section 12.3](#), and an approach to the specification of access patterns supporting tunable load and locality conditions in [Section 12.4](#), all the more reason for an imminent end to this introductory blather.

12.1 Notation

A few ideas to make life easier in this chapter that have not been needed previously, such as scalar multiplication and probability distributions, are worth a short explanation.

12.1.1 Scalar multiplication

For lists $x \in \mathbb{R}^*$ of real numbers and scalars $k \in \mathbb{R}$, the notation

$$k \cdot x = x \cdot k = (\lambda t. kt)^* x \quad (12.1)$$

refers to the list obtained from multiplying every term of x by k . For example, $2 \cdot \langle 3, 4, 5 \rangle$ has a value of $\langle 6, 8, 10 \rangle$. This notation extends to lists of lists $z \in \mathbb{R}^{**}$ in the obvious way.

$$k \cdot z = z \cdot k = (\lambda x. k \cdot x)^* z$$

12.1.2 Permutations

For any list $x \in S^*$, we write $\wp x \in \mathcal{P}(S^*)$ for the set of all lists derivable from x by rearranging its terms in any order.

$$\wp x = (\mu \lambda p. x \circ p) (\mathbb{N}^{|x|} \cap (\mathcal{D}(x) \twoheadrightarrow \mathcal{D}(x))) \quad (12.2)$$

The expression $\mathbb{N}^{|x|} \cap (\mathcal{D}(x) \twoheadrightarrow \mathcal{D}(x))$ refers to the set of all permutations of length $|x|$ as usual. For example, $\wp \langle a, b, c \rangle$ has a value of

$$\wp \langle a, b, c \rangle = \{ \langle a, b, c \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle a, c, b \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle \}.$$

12.1.3 Zipped function application

For a list of functions $f \in (S \rightarrow T)^*$ and a list of arguments $x \in S^*$, the expression $f \triangle x$ denotes the list of results $y \in T^*$ obtained by applying each function in f to the corresponding argument in x .

$$f \triangle x = (\lambda i. f_i x_i)^* \iota_{\min\{|f|, |x|\}} \quad (12.3)$$

For example, $\langle \lambda a. a + 1, \lambda b. b^2, \lambda c. \sqrt{c} \rangle \triangle \langle 2, 3, 4 \rangle$ has a value of $\langle 3, 9, 2 \rangle$. Extra functions or arguments are allowed and any surplus on either side is ignored.

12.1.4 Probability theory

Some familiarity with basic probability theory would be useful for an understanding of arbiter performance analysis. The introductions in any of [27, 101, 159] contain more than enough background for this chapter. The rest of this section is a brief survival guide covering the bare minimum.

Probability

An informal operational definition of the **probability** of an event, as in “a grant appears on the i -th terminal with probability p ”, is the fraction $0 \leq p \leq 1$ of experimental trials toward which that event tends to be observed over the course of sufficiently many repetitions.¹ Hereafter we use the conventional notation

$$[0, 1] = \{p \in \mathbb{R} \mid 0 \leq p \wedge p \leq 1\}$$

for the set of real numbers in this range.

Conditional probability

The related concept of **conditional probability** pertains to two events. For events denoted a and b , the conditional probability of $a \mid b$, read “ a given b ”, would correspond in the current context to the number of trials in which both a and b occur divided by the number of trials in which b occurs with or without a , under the continued assumption of a large number of trials.

- The conditional probability of $a \mid b$ is not uniquely determined by the total probabilities of a and b , and need not match that of $b \mid a$, but the conditional probability of $a \mid a$ is always 1.
- If the conditional probabilities of $a \mid b$ and $b \mid a$ are zero, then the events a and b are said to be **mutually exclusive**, meaning if either of them happens, the other can not.
- If the conditional probability of $a \mid b$ matches the total probability of a , and likewise the conditional probability of $b \mid a$ matches the total probability of b , then the events a and b are said to be **independent**, meaning the occurrence of one implies nothing about the other.

¹This concept of probability, known as the “frequentist” interpretation [278], has long been superseded by more rigorous alternatives, but is adequate on an intuitive level for a survival guide like this one. Do not cite this definition on a math test or in a job interview.

Joint probabilities

However abstract they may seem at the moment, a couple of rules of thumb become important when we want to estimate or calculate the probability of a chain of events.

- If an event is deemed to have occurred precisely when either of two mutually exclusive events with respective probabilities p_0 and p_1 occurs, then its probability is necessarily $p_0 + p_1$.
- An event deemed to have occurred if and only if both of two independent events with respective probabilities p_0 and p_1 occur necessarily has a probability equal to the product $p_0 p_1$.

Distributions

A common setting features the possibility of n mutually exclusive events of which it is known *a priori* that exactly one must occur, for example when one grant must issue from an n -input arbiter subject to n concurrent requests. Being both mutually exclusive and exhaustive, these events must have probabilities $p = \langle p_0 \dots p_{n-1} \rangle \in [0, 1]^n$ satisfying $\sum p = 1$. The list $p \in [0, 1]^n$ regarded as a function in this situation is called a **distribution**.

Expectation

If a quantifiable result $x_i \in \mathbb{R}$ (such as the latency of a grant) is obtained whenever the i -th of n mutually exclusive events occurs, and these events are described by a distribution $p \in [0, 1]^n$, then the **expectation** of x is defined as

$$\sum_{i=0}^{n-1} p_i x_i = \sum \Pi^* \langle p, x \rangle^T$$

also called the **mean** or average. Expectation in this context is a technical term, not necessarily what anyone literally expects. (For example, no one expects a score of $3\frac{1}{2}$ on a dice roll.) See [Section 11.1.2](#) for a reminder about the transpose notation.



12.2 Arbiter decompositions

There are at least three distinct ways of putting arbiters together before even contemplating all the ways to mix them up. Arbiters in the form of a **mesh** are one alternative [120], trees are another, and the rest take the form of a **token ring**. Each of these is the topic of one of the next few sections.

12.2.1 Mesh

A mesh arbiter with n inputs consists of several cascaded stages each having n inputs and n outputs. Each stage partitions the inputs into several subsets so that each signal competes against others within the same subset. The connections between stages are permuted to shuffle the subsets. No two signals that have competed against each other in previous stages meet again in any subsequent stage, but every signal gets a chance in some stage to compete with any other signal that has not otherwise been eliminated.

An example of a mesh arbiter with six inputs is shown in [Figure 12.1](#). A way of seeing that it meets the condition explained above is to trace the path from an input labeled by any letter to

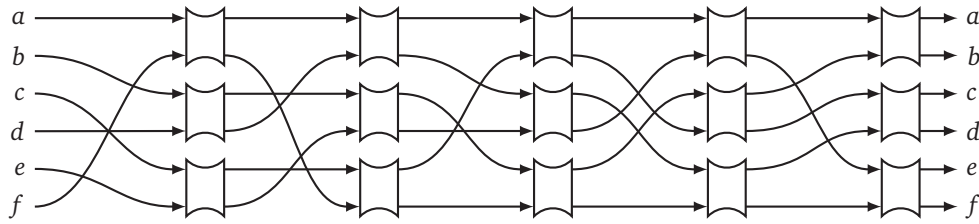


Figure 12.1: A six-way square mesh arbiter has five stages and three ARB primitives in each stage.

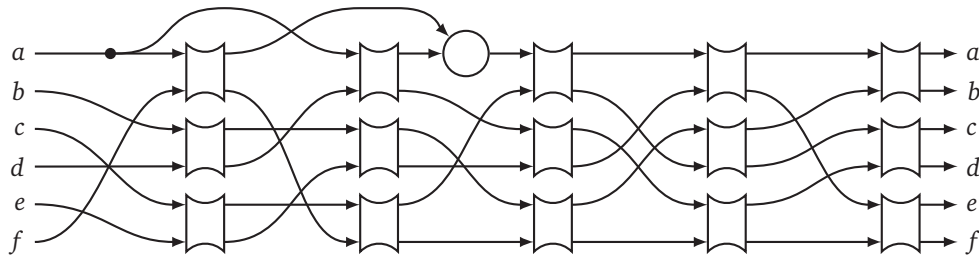


Figure 12.2: Broadcasting one of the signals to the first two stages shortens its critical path.

an output labeled by the same letter, confirming that it passes through the same ARB primitive as that of any other letter exactly once. Less evident from this example is that an arbiter of this form generalizes to any number of inputs and can be built from arbiters of varying arities in place of the primitives.

A disadvantage of the mesh arbiter is the latency incurred by the large number of arbiters required in series [194]. Is there any way of speeding it up? If a small additional hardware cost is acceptable, one remedy is to broadcast a signal to multiple stages as shown in Figure 12.2. The input labeled *a* still competes with every other by passing through five ARB primitives, but conceivably takes less time than five ARB delays to emerge at the output by competing with *d* and *f* simultaneously if necessary.

If some concurrency is good then maybe more is better. Instead of broadcasting just one of the signals to the first two stages, we could broadcast every signal to every stage. As an extra benefit, such a design would be fairer because it would not favor any one signal over the others. The only problem is that the resulting circuit would no longer be an arbiter because the output stage could emit three concurrent grants in response to six requests, having received all six directly from the proposed broadcast network including those blocked by other stages.

Less concurrent but less wrong would be to broadcast every signal to the first two stages and let them propagate sequentially as normal through the rest. This more conservative approach guarantees mutual exclusion, but unfortunately suffers from deadlock. If there are six concurrent requests, with those labeled *a*, *b*, and *c* competing successfully against their respective alternatives in the first stage, and the other three prevailing in the second stage, then no further progress is possible because each of the six JOIN primitives driving the third stage will have received a signal on one of its inputs and not the other.

Nevertheless, [Figure 12.2](#) is not the only solution. A more concurrent middle ground achieving both mutual exclusion and freedom from deadlock is possible by staggering the broadcast zones separately for each signal, but only in more complicated ways than should be attempted manually. Developing an algorithm to generate precisely the valid combinations of broadcast zones for a given mesh arbiter turns out to be the relatively easy part. Constructing the family of mesh arbiters for a given arity is the first order of business.

Mesh enumeration

If we temporarily ignore the ordering of the stages in a mesh, then it is determined by a set of stages where each stage is determined by a set of arbiters and each arbiter is determined by the set of signals it is wired to receive. If each signal is identified with one of a consecutive sequence of natural numbers starting from zero, then everything there is to know about the mesh is summarized by a set $k \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$. For a given arity $n \in \mathbb{N}$, we would know all meshes having n inputs if some function $B_0 : \mathbb{N} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))))$ were known for which $B_0 n \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))))$ contains all values of k corresponding to them.

To construct B_0 , we can start by formalizing the conditions a mesh must satisfy as constraints on the set k . The set of signals associated with each arbiter in a stage must be disjoint from the sets associated with other arbiters in the same stage, but collectively they form a partition on the set of signals. Letting $u t \in \mathcal{P}(\mathcal{P}(\mathcal{P}(s)))$ denote the set of all partitions on a set t based on a definition

$$u = \lambda t. \bigcup_{f \in t \rightarrow t} \{(\pi f) t\}$$

and [Equation 6.6](#) means one thing to be said about a set k describing a mesh with arity n is that it is a member of $\mathcal{P}(u \mathcal{R}(t_n))$.

Another constraint on the mesh implies a requirement for any two signals $i, j \in \mathcal{R}(t_n)$ to be common to exactly one class $c \in p$ of exactly one partition $p \in k$. Letting $v p$ denote the set of all unordered pairs of signals associated with each other by any class in a partition p based on the definition

$$v = \lambda p. \bigcup_{c \in p} \bigcup_{i \in c} \bigcup_{j \in c - \{i\}} \{\{i, j\}\} \quad (12.4)$$

allows us to write $(\mu v) k$ by [Equation 5.1](#) for the set of all sets of unordered pairs of signals in competition with each other at any point throughout the mesh. The requirement that any two signals have exactly one opportunity to compete with each other is then equivalent to the condition

$$(\mu v) k \in u v \{\mathcal{R}(t_n)\}$$

meaning the sets of unordered pairs of competitors per stage form a partition themselves on the set of possible unordered pairs of competitors over the full range of signals. These conditions suggest the following definition for the desired function B_0 .

$$B_0 = \lambda n. (\lambda(u, v). \bigcup_{k \in \mathcal{P}(u \mathcal{R}(t_n))} (\lambda l. \langle \emptyset, \{k\} \rangle_l \delta_{u v \{\mathcal{R}(t_n)\}}^{\{(\mu v) k\} \cup u v \{\mathcal{R}(t_n)\}})) (\lambda t. \bigcup_{f \in t \rightarrow t} \{(\pi f) t\}, \lambda p. \bigcup_{c \in p} \bigcup_{i \in c} \bigcup_{j \in c - \{i\}} \{\{i, j\}\}))$$

Smarter mesh enumeration

Unfortunately, any computation based naively on this definition of B_0 is absolutely infeasible for all but miniscule values of n , so we have to look for better algorithms, useful approximations, or both.

- One way of approximating this function without missing anything important is to exclude most of the results that give rise to the same meshes as others because they differ only in the way the signals are numbered.
- Another approximation excludes any results k having a stage described by a partition $p \in k$ in which every class $c \in p$ contains only one signal, because such a stage is only a bus.
- With regard to computational shortcuts, some partitions $p \in k$ describing mesh stages may contain some sets $c \in p$ with $|c| = 1$, especially when the arity n is odd, but there is no need to record the unit sets explicitly because a subset $p' \subseteq p$ excluding them would enable the recovery of p based on $\mathcal{R}(t_n) - \bigcup p'$.
- Similarly, some partitions $p \in k$ may contain a single set $c \in p$ with $|c| = 2$ and the rest unit sets. Excluding these partitions from a subset $k' \subseteq k$ would still allow k to be inferred easily from $\mathcal{R}(t_n) - \bigcup \bigcup k'$.

More ideas follow shortly, but these are an adequate start. Algorithms for generating partitions of a set are well known and not hard to improvise [146], so we omit any constructive definition and assume a set $B_1 n \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$ of subsets of partitions on $\mathcal{R}(t_n)$ specified as

$$B_1 = \lambda n. ((\mu \lambda r. \{c \in (\pi r) \mathcal{R}(t_n) \mid 1 < |c| \wedge |c| < n\}) \mathcal{R}(t_n)^n) - \{p \in \mathcal{P}(\mathcal{P}(\mathbb{N})) \mid 2 \geq |\bigcup p|\}. \quad (12.5)$$

As noted above, $B_1 n$ excludes classes of c of the form $c = \{i_0\}$ from the partitions, and also excludes partitions of the form $p = \{\{i_0, i_1\}\}$ and $p = \emptyset$.

Next, to formalize the notion of equivalence among meshes with respect to renumbering, let the **shape** of a stage $t \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ in a mesh $m \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$ be understood as the list

$$(\lambda(u, c). u)^* ((\mu \lambda c. (|c|, c)) t)^{\circ-1} \in \mathbb{N}^*$$

of the arities $|c|$ of its arbiters in ascending order, and the shape of a mesh m as the list $B_2 m \in \mathbb{N}^{**}$ of shapes of its stages in ascending lexicographic order, with B_2 defined as

$$B_2 = (\lambda l. \lambda t. (\lambda(u, i). u)^* ((\mu \lambda i. (|i|, i)) t)^{\circ-1})^2 \lambda c. |c|.$$

Then two sets $m_0, m_1 \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$ determine the same mesh whenever $B_2 m_0$ is equal to $B_2 m_1$, so we need not seek more than one set for each shape.

A reasonable way forward is to phrase the problem in terms of graph theory. A partition subset $p \in B_1 n$ representing a stage in a mesh corresponds to a vertex in an undirected graph whose edges connect any two vertices p and q satisfying $(\nu p) \cap \nu q \neq \emptyset$ by Equation 12.4, which is to say that two of the same signals compete with each other somewhere in both stages. The adjacency set $B_3(p, t)$ of an arbitrary vertex p in a graph g with vertices $t = \mathcal{D}(g) = B_1 n$ under this interpretation is given by

$$B_3 = \lambda(p, t). (\lambda \nu. \{q \in t - \{p\} \mid (\nu p) \cap \nu q \neq \emptyset\}) \lambda p. \bigcup_{c \in p} \bigcup_{i \in c} \bigcup_{j \in c - \{i\}} \{\{i, j\}\} \quad (12.6)$$

and the whole graph g therefore by

$$g = \prod_{h \in t} \{h\} \times B_3(h, t)$$



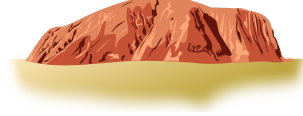
with notation defined in Equation 6.7. A valid mesh should never have the same two signals competing with each other in multiple stages, so the vertices describing its stages should be pairwise non-adjacent in the graph, forming what is known as an **independent set**. If the set

$$i = \{s \in \mathcal{P}(g) \mid \mathcal{D}(s) \cap \bigcup \mathcal{R}(s) = \emptyset\} \quad (12.7)$$

of all independent sets induced by the graph g were known, it would be only a small step further to write

$$(\mu \lambda u. \min u) (\pi B_2) (\mu \lambda s. \mathcal{D}(s)) i$$

for the subset of materially distinct mesh specifications.



A hard place

Solving Equation 12.7 entails finding the maximum independent sets (among others), which is a notorious NP-hard problem, but valiant efforts to game it abound in the literature. Along with well studied heuristics [9, 88, 170, 208], efficient algorithms are known for restricted families of graphs, most notably **chordal** graphs [84, 96, 108] and their generalizations [87].

Of possible interest in the current setting would be the minor restriction to meshes m whose stages are identically shaped (that is, with $|\mathcal{R}(B_2 m)| = 1$, meaning the arbiters have similar arities in every stage). This restriction would procure **vertex transitive** graphs, so that obtaining just one independent set would yield all others of the same cardinality through permutations of the signal numbers. The graphs would also benefit from a tight (but not exact) upper bound of

$$\frac{n(n-1)}{2(vp)}$$

on the maximum independent set size, with n being the number of input signals, $p \in \mathcal{D}(g)$ being any vertex in the graph, and v given by Equation 12.4, because there are $n(n-1)/2$ unordered pairs of competing signals, and $v p$ competitions taking place in each stage. Knowing an upper bound on the maximum independent set size saves time in a heuristic search by providing a stopping criterion, and this upper bound beats the more general results in [48, 99, 154, 168] for graphs of comparable sizes. Furthermore, graphs following from this restriction can often be chordal, as easily checked by a **lexicographic breadth-first search** [257, 280], making their maximum independent set problem efficiently solvable as noted above.

Mesh enumeration concluded

Computing the exact solution to Equation 12.7 with certainty in the unrestricted case takes exponential time but can be made relatively space efficient by a recurrence that does not require enumerating the power set of the graph in advance. For any non-empty list $h : t = (B_1 n)^{\circ-1}$ of subsets of partitions on signal numbers, there is always one independent set $\{h\}$. There may be other independent sets $s \in \mathcal{P}(\mathcal{R}(t))$ built from the tail of the list $h : t$, and there may be still others of the form $m \cup \{h\}$ where m is an independent set satisfying

$$m \in \mathcal{P}(\mathcal{R}(t \uparrow \mathcal{R}(t) - B_3(h, \mathcal{R}(t))))$$

meaning m is inferred from a list derived from t with terms adjacent to h left out by Equation 12.6. These observations suggest that a function $B_4 : \mathcal{P}(\mathcal{P}(\mathbb{N}))^* \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))))$ satisfying the

recurrence

$$B_4(b) = \langle (\lambda(h : t). (\mu \lambda u. \min u) (\pi B_2) ((B_4 t) \cup \{\{h\}\} \cup \bigcup_{m \in B_4(t \uparrow \mathcal{R}(t) - B_3(h, \mathcal{R}(t)))} \{m \cup \{h\}\})) b, \emptyset \rangle_{\delta^\epsilon} \quad (12.8)$$

would make the full complement of distinct mesh specifications expressible as $B_4 (B_1 n)^{\circ-1}$, where redundant results are suppressed by $(\mu \lambda u. \min u) \circ \pi B_2$ as noted previously. This formulation does not rely on the argument b being lexicographically ordered, and certain other orderings may be conducive to small additional time efficiencies.²

A fitting conclusion depends on two conditions previously deferred: each $k \in B_4 (B_1 n)^{\circ-1}$ needs to be transformed to a member of $B_0 n$ having precisely the interpretation proposed on page 366, and the omitted triangular mesh needs to be supplied explicitly in the result. The first condition calls for a stage containing $\{i, j\}$ to be added to k for every two signals $i, j \in \mathcal{R}(t_n)$ not explicitly appearing together in some extant class $c \in \bigcup k$. In terms of the function v defined by Equation 12.4,

$$k \cup \bigcup_{c \in v\{\mathcal{R}(t_n)\} - v \bigcup k} \{\{c\}\}$$

expresses a set derived from k extended with the necessary additional stages. Consistency with $B_0 n$ as originally conceived also requires any stage $l \in k$ lacking any members of $\mathcal{R}(t_n)$ in its classes $c \in l$ to be padded with unit sets to make up the difference. That is, the stage l should be transformed to

$$l \cup \bigcup_{i \in \mathcal{R}(t_n) - \bigcup l} \{\{i\}\}$$

Both of these transformations to all relevant sets k would be summarized conveniently by a function

$$B_5 = \lambda n. (\lambda v. (\mu^2 \lambda l. l \cup \bigcup_{i \in \mathcal{R}(t_n) - \bigcup l} \{\{i\}\}) \circ (\mu \lambda k. k \cup \bigcup_{c \in v\{\mathcal{R}(t_n)\} - v \bigcup k} \{\{c\}\}) \lambda p. \bigcup_{c \in p} \bigcup_{i \in c} \bigcup_{j \in c - \{i\}} \{\{i, j\}\})$$

in the context $(B_5 n) B_4 (B_1 n)^{\circ-1}$ were it not for the second condition noted above.

With regard to this latter condition, the so called triangular mesh arbiter of arity n features $n(n-1)/2$ stages each containing a primitive two-input arbiter for two of the signals and wires to carry the other $n-2$. For an arity of $n=3$, there is no other alternative. However, in the interests of both mathematical simplicity and computational efficiency, $B_4 (B_1 n)^{\circ-1}$ as defined omits this mesh, obliging us to insert it manually. Fortunately, it emerges naturally from nothing as $(B_5 n) \{\emptyset\}$, so that the complete set of explicit mesh specifications follows as $B_6 n$ with

$$B_6 = \lambda n. (B_5 n) (\{\emptyset\} \cup B_4 (B_1 n)^{\circ-1})$$

taken for a less redundant and computationally simpler compatible replacement version of B_0 .

Broadcast zone enumeration

With mesh enumeration somewhat settled, we return to the question of selecting broadcast zones deferred since page 366. That is, we seek ways to save time by transmitting the same signals to

²Sorting by shapes can help. As a spot check for readers interested in replicating this result, the author's implementation reports 0, 4, 17, 131, and 898 distinct meshes respectively for arities n ranging from 3 to 7. In strict adherence to Equation 12.5 and Equation 12.8, these figures exclude the implicit triangular mesh of each arity.

multiple stages of a mesh concurrently, but without impairing its operation. The plan is for each signal to have its own independent sequence of broadcast zones, with each zone covering one or more consecutive stages of the mesh. The outputs of the same signal from each stage of a zone synchronize in a JOIN and then are broadcast through a FORK to the inputs for that signal in every stage of the next zone, and so on. Because the signal propagates through the stages in each zone concurrently and only through consecutive zones sequentially, and because there may be fewer zones than stages, the signal might have the opportunity to finish faster than it would otherwise.

Further progress in this task requires fixing an order among the stages by modeling a mesh m henceforth as a list of stages instead of a set, with m_0 representing the front end stage and $m_{|m|-1}$ the back. Two meshes having the same stages in different orders are regarded as distinct, and this distinction is reasonable because they might perform differently or permit different broadcast networks. While there is no comparable reason to regard one arbiter as preceding another within the same stage, it is technically convenient to make a list of them as well, with the lexicographic ordering by signal numbers as good as any. A specification $m \in \mathcal{P}(\mathbb{N})^{**}$ of the form now under consideration can be obtained from

$$m \in \wp (\lambda l. l^{\circ-1})^* k^{\circ-1} \quad (12.9)$$

by Equation 12.2 for any $k \in B_6 n$ previously derived.

When there are n inputs, the broadcast network can be described by a parameter $b \in \mathbb{N}^{*n}$, with the broadcast zones for the i -th signal described by $b_i \in \mathbb{N}^*$ for $0 \leq i < n$. Each term b_{ij} records the number of stages covered by the j -th zone of the i -th signal. For example, if for signal number 3 we have $b_3 = \langle b_{30}, b_{31}, b_{32} \rangle = \langle 2, 3, 1 \rangle$, then that signal has three broadcast zones covering 2, 3, and 1 stages respectively, with the first zone covering stages 0 and 1, the next covering stages 2 through 4, and the last covering stage 5. Other signals in the same mesh might have a different number of zones or have zones of different sizes, but clearly a mesh m with n inputs is compatible only with broadcast networks b satisfying $\sum b_i = |m|$ for all $0 \leq i < n$.

However, even among those meeting this condition, not all broadcast networks b are valid for a mesh m . A wrong choice of b could enable deadlock or could compromise mutual exclusion. Short of building the circuit or simulating it as a Petri net, how might we divine the validity of a broadcast network in advance? We can start by envisioning all n requests being made simultaneously to the mesh. Whatever broadcast zones there are and whatever happens in the rest of the stages, we have the set

$$(m_0)^{\mathsf{T}} \in \mathcal{P}(\mathbb{N}^{|m_0|})$$

of possible lists of signals that can pass through the first stage, where each $\langle s_0, s_1 \dots s_{|m_0|-1} \rangle \in (m_0)^{\mathsf{T}}$ contains a signal $s_0 \in m_{00}$ from the first arbiter in the stage, $s_1 \in m_{01}$ from the next, and so on, with always exactly one signal from each arbiter (Section 11.1.2).

If it were known by psychic power that a particular list of signals $s \in (m_0)^{\mathsf{T}}$ were destined to pass the first stage, then we could simplify the analysis of subsequent stages by excluding the signals blocked in the first stage due to s , which would be $K_0(m_0, s) \in \mathbb{N}^*$ in terms of a function

$$K_0 = \lambda(m, s). ((\bigcup \mathcal{R}(m)) - \mathcal{R}(s))^{\circ-1}$$

that makes a list of everything left over when terms of s are removed from those of m_0 . Excluding these signals means rewriting m to a smaller mesh with the signals deleted from subsequent stages,



and then treating the latter as the whole mesh for purposes of further analysis. However, because some signals $h \in \mathcal{R}(K_0(m_0, s))$ might have a large initial broadcast zone $b_{h0} > 1$, they should be deleted only from stages $m \ll b_{h0}$ beyond the zone and retained in the initial stages $m \vdash b_{h0}$, to which h is broadcast regardless. Accordingly for a single h , the mesh m is rewritten to

$$(m \vdash b_{h0}) \parallel (\lambda c. c - \{h\})^{**} (m \ll b_{h0})$$

but to account for all of $K_0(m_0, s)$ we need

$$(\mathcal{F}_m \lambda(h, t). (t \vdash b_{h0}) \parallel (\lambda c. c - \{h\})^{**} (t \ll b_{h0})) K_0(m_0, s)$$

as the rewritten mesh, or even more specifically

$$m' = ((\mathcal{F}_m \lambda(h, t). (t \vdash b_{h0}) \parallel (\lambda c. c - \{h\})^{**} (t \ll b_{h0})) K_0(m_0, s)) \ll 1 \quad (12.10)$$

for the rewritten subsequent stages of the mesh.

Because the actual list of signals $s \in (m_0)^\top$ passing the first stage is unpredictable, a realistic solution to this problem entails generalizing from this fanciful hypothetical case to all possible instances of s and the corresponding reduced meshes m' , preferably organized manageably. A directed graph with edges labeled by lists s is a good start. Each node in the graph is a pair (b', m') describing a broadcast network b' and a mesh m' reduced as in Equation 12.10 relative to its antecedent node (b, m) according to the list s labeling the edge connecting them. A reduced broadcast network b' reflecting the removal of the first stage would be $b' = K_1 b$ with

$$K_1 = (\lambda z. \langle (\lambda(h : t). \langle h - 1 : t, t \rangle_{\delta_1^h}) z, \epsilon \rangle_{\delta_\epsilon})^*$$

being the function that either decrements the size of every first broadcast zone or deletes it.

Enumerating the tuples $(s, (b', m'))$ for all $s \in (m_0)^\top$ with respect to a pair (b, m) would yield the adjacency set of the node (b, m) in the graph for the most part, but to summarize this operation in general with respect to an arbitrary node, we should take into account that m may have been reduced previously, and so could be empty or could have stages containing empty sets. To avoid anomalous results in these cases, let the edge label s be drawn from the set

$$((m \parallel \langle \epsilon \rangle)_0 \uparrow \mathcal{P}(\mathbb{N}) - \{\emptyset\})^\top - \{\epsilon\}$$

as a more robust alternative to $(m_0)^\top$, which is empty rather than undefined when m is empty, ignores empty members of m_0 , and excludes $s = \epsilon$. Then with the adjacency set $K_2(b, m)$ given by

$$K_2 = \lambda(b, m). \bigcup_{s \in ((m \parallel \langle \epsilon \rangle)_0 \uparrow \mathcal{P}(\mathbb{N}) - \{\emptyset\})^\top - \{\epsilon\}} \{(s, (K_1 b, ((\mathcal{F}_m \lambda(h, t). (t \vdash b_{h0}) \parallel (\lambda c. c - \{h\})^{**} (t \ll b_{h0})) K_0(m_0, s)) \ll 1))\}$$

we are well on the way to building a graph $g = K_3(b, m)$ of nodes (v, e) with edges $(s, v') \in e$ labeled by lists of signals $s \in \mathcal{R}(t_n)^*$ based on

$$K_3 = \lambda(b, m). \Gamma_{(b, m)} (\mu \lambda v. (v, K_2 v)) ((\mathbb{N} - \{0\})^{**} \times \mathcal{P}(\mathbb{N})^{**})$$

and notation defined in Equation 6.5.

The payoff from building this graph is that it answers the question of whether a proposed list b describes a valid broadcast network for a given mesh m . There may be several ways to characterize

this validity. If there is no deadlock, then every path from the initial node (b, m) to a terminal node has a length $|m|$ equal to the number of stages, because progress is always possible at each stage. If in addition mutual exclusion is preserved, then every edge leading to a terminal node is labeled by a unit list, because there is only one signal the back end stage can emit. A simple condition that appears to subsume both requirements is to have precisely $\mathcal{R}(\iota_n)^1$ as the set of incident edge labels on terminal nodes. With respect to the graph $g = K_3(b, m)$, we may write $\bigcup \mathcal{R}(g)$ for the set of edges, $\mathcal{D}(g \cap (\mathcal{D}(g) \times \{\emptyset\}))$ for the set of terminal nodes, and $K_4 g$ for the set of labels on edges leading to terminal nodes in terms of

$$K_4 = \lambda g. \mathcal{D}((\mathbb{N}^* \times \mathcal{D}(g \cap (\mathcal{D}(g) \times \{\emptyset\}))) \cap \bigcup \mathcal{R}(g)).$$

Then for a given $m \in \mathcal{P}(\mathbb{N})^{**}$ describing a mesh of arity n , the set of valid broadcast zone specifications $b \in \mathbb{N}^{**}$ is $K_5(n, m)$ for K_5 defined as follows.

$$K_5 = \lambda(n, m). \{b \in (\mathbb{N} - \{0\})^{*n} \mid \sum^* b = |m|^{\#} \wedge K_4 K_3(b, m) = \mathcal{R}(\iota_n)^1\} \quad (12.11)$$

This analysis can confirm, for example, that there are no valid broadcast networks for the mesh in Figure 12.1 with any zones covering more than three stages, but several valid networks admit sporadic three-way zones, such as

$$\langle\langle 3, 1, 1 \rangle, \langle 1, 1, 1, 1, 1 \rangle, \langle 1, 3, 1 \rangle, \langle 3, 1, 1 \rangle, \langle 1, 3, 1 \rangle, \langle 1, 1, 1, 1, 1 \rangle\rangle$$

under the assumption of signals being numbered alphabetically. A few mixtures of two-way and three-way zones are also valid, such as the following.

$$\langle\langle 1, 3, 1 \rangle, \langle 1, 3, 1 \rangle, \langle 1, 3, 1 \rangle, \langle 1, 2, 1, 1 \rangle, \langle 1, 3, 1 \rangle, \langle 1, 2, 1, 1 \rangle\rangle$$

The latter cuts down the critical paths to at most four arbiters in series from the original five.

Mesh decompositions

The space of mesh arbiter decompositions includes a base case not represented by any $k \in B_6 n$, namely the single-input arbiter implemented by a wire 1. The single-input arbiter can be included specially in a definition of

$$\nabla_u : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**})$$

taking a positive arity $n \in \mathbb{N}$ to the set of pairs $(b, m) \in \mathbb{N}^{*n} \times \mathcal{P}(\mathbb{N})^{**}$ describing arbiters with arity n , broadcast networks $b \in \mathbb{N}^{*n}$, and meshes $m \in \mathcal{P}(\mathbb{N})^{**}$, which we construct now for future reference. A pair (b, m) pertaining to the base case arbiter has exactly $|m| = 1$ stage with $|m_0| = 1$ arbiter in the stage, the sole broadcast zone $b_{00} = 1$ and the unit set $m_{00} = \{0\}$, implying

$$(b, m) = (\langle\langle 1 \rangle\rangle, \langle\langle \{0\} \rangle\rangle).$$

For larger arities n , we have m given by Equation 12.9 and b being any member of $K_5(n, m)$ by Equation 12.11, suggesting the expression

$$\bigcup_{k \in B_6 n} \bigcup_{m \in \mathcal{P}(\lambda l. l^{\circ-1})^* k^{\circ-1}} K_5(n, m) \times \{m\}$$

for the set of all pairs (b, m) , and hence the following definition for ∇_u overall.

$$\nabla_u = \lambda n. \langle\emptyset, \{\langle\langle 1 \rangle\rangle, \langle\langle \{0\} \rangle\rangle\}\rangle_{\delta_1^n} \cup \bigcup_{k \in B_6 n} \bigcup_{m \in \mathcal{P}(\lambda l. l^{\circ-1})^* k^{\circ-1}} K_5(n, m) \times \{m\} \quad (12.12)$$



Mesh arbiter combining form

When a specification $(b, m) \in \nabla_u n$ for a mesh arbiter of arity n is decided and the list $x \in \mathbb{H}^{|b| m|}$ of its constituent arbiters is ready, combining them into the specified mesh arbiter $\Omega_u((b, m), x) \in \mathbb{H}$ is the next step. A description of this transformation in general terms calls for a definition of the combining form.

$$\Omega_u : (\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}) \times \mathbb{H}^* \rightarrow \mathbb{H}$$

This construction follows the convention that the first $|m_0|$ arbiters in x are for the front end stage, the next $|m_1|$ are for the succeeding stage, and so on to the last $|m_{|m|-1}|$ terms of x forming the back end stage. Furthermore, the first arbiter associated with the i -th stage for any i arbitrates among signals identified by m_{i0} , the next with m_{i1} , and so on throughout the stage. Hence we have the i -th stage consisting of arbiters in the sublist

$$x \circ \iota_{|m_i|}^{|b(m_{i1})|} \in \mathbb{H}^*$$

of x , which can be used to express the block

$$(\overset{\circ}{b} m_i)^{-1} \times (\mathcal{F} \mathbf{R}) (x \circ \iota_{|m_i|}^{|b(m_{i1})|}) \times \overset{\circ}{b} m_i \in \mathbb{H}$$

having its $n = |\overset{\circ}{b} m_i|$ signals permuted into ascending numerical order on the input and output sides. (See Equation 11.2 for a reminder about this notation.) It is only a short step further to express the full block of stages as

$$(\mathcal{F} \mathbf{R}) (\lambda i. (\overset{\circ}{b} m_i)^{-1} \times (\mathcal{F} \mathbf{R}) (x \circ \iota_{|m_i|}^{|b(m_{i1})|}) \times \overset{\circ}{b} m_i)^* \iota_{|m|}$$

with the inputs and outputs both grouped into n buses of $|m|$ lines each in $D_0(m, x)$ by

$$D_0 = \lambda(m, x). (\lambda p. p \times (\mathcal{F} \mathbf{R}) (\lambda i. (\overset{\circ}{b} m_i)^{-1} \times (\mathcal{F} \mathbf{R}) (x \circ \iota_{|m_i|}^{|b(m_{i1})|}) \times \overset{\circ}{b} m_i)^* \iota_{|m|} \times p^{-1}) \iota_{|m|} \overset{\circ}{b} m_0 \times |m|.$$

Before connecting anything to anything else in this block, we have to contemplate the broadcast network described by b . For any $0 \leq i < n$, every zone size $h \in \mathcal{R}(b_i)$ in the list $b_i \in \mathbb{N}^*$ of zone sizes for the i -th signal calls for a network **FORK** h feeding into certain arbiters across h consecutive stages, and a network **JOIN** h collecting the same signals from the other side of the same arbiters. Then the output from the **JOIN** network in each zone needs to be connected to the **FORK** input on the next. Attending to these latter connections first can be done with a cascade

$$(\mathcal{F}_1 \lambda(h, t). \mathbf{L}\langle \mathbf{R}(\mathbf{FORK} h, \mathbf{JOIN} h), t \rangle \uparrow 1) b_i$$

taking care to connect the last output from $\mathbf{R}(\mathbf{FORK} h, \mathbf{JOIN} h)$ in each zone to the last input of the result t from the rest of the zones, which is really the first input to the next zone due to the inputs of the block $\mathbf{L}\langle \mathbf{R}(\mathbf{FORK} h, \mathbf{JOIN} h), t \rangle$ being rolled up on each iteration. This operation applied to every zone results in a list of blocks

$$z = (\mathcal{F}_1 \lambda(h, t). \mathbf{L}\langle \mathbf{R}(\mathbf{FORK} h, \mathbf{JOIN} h), t \rangle \uparrow 1)^* b$$

in which each block z_i has $|m| + 1$ input terminals and $|m| + 1$ outputs, with the last input and output meant to be exposed to the environment and the rest meant to interface between the stages of the mesh. The terminals for interfacing between stages are organized into $|b_i|$ buses with one

input and one output bus for each zone, the j -th bus of each having b_{ij} lines. A block combining the whole broadcast network in a way that collects the intended external terminals at the end in reverse order of signal numbers is given by $D_1 b$ with $D_1 : \mathbb{N}^{**} \rightarrow \mathbb{H}$ defined as

$$D_1 = (\mathcal{F}_{Z_1} \lambda(h, t) \cdot \mathbf{R}(h \downarrow 1, t) \uparrow 1) \circ (\mathcal{F}_1 \lambda(h, t) \cdot \mathbf{L}\langle \mathbf{R}(\text{FORK } h, \text{JOIN } h), t \rangle \uparrow 1)^*$$

where it should be noted that the rest of the terminals are ordered normally. That is, the first $|m|$ outputs from $D_1 b$ should be connected to the inputs associated with the first signal on arbiters throughout the mesh, the next $|m|$ with the second signal, and so on, while the first $|m|$ inputs to $D_1 b$ should come from the outputs of the first signal throughout the mesh, and so on.

The block $D_0(m, x)$ as defined above enables all connections from outputs of the broadcast network to inputs of the mesh in the simple expression

$$\mathbf{F}_t \langle D_1 b, D_0(m, x) \rangle$$

for a total number of bus lines $t = n|m| = \sum b$. The result exposes $t + n$ inputs on the broadcast network, n outputs on the broadcast network, and t outputs on the mesh in that order, of which the last still need connecting to the first t inputs on the broadcast network in the same order. To this end, we route the latter outputs through a bus l^t

$$\mathbf{Z}^t \mathbf{R}(\mathbf{F}_t \langle D_1 b, D_0(m, x) \rangle \downarrow t, l^t)$$

which temporarily reverses their order, and then wrap the bus outputs around to the front

$$\mathbf{Z}^t ((\mathbf{Z}^t \mathbf{R}(\mathbf{F}_t \langle D_1 b, D_0(m, x) \rangle \downarrow t, l^t)) \uparrow t)$$

which restores it. Although this expression leaves the n external inputs in the reverse order with respect to the signal numbering scheme, correctness is unaffected because the outputs are similarly ordered.

This expression could constitute the whole definition of $\Omega_u((b, m), x)$, or we could make it a tiny bit smarter by having it infer one of the two possible results regardless of x whenever $|m| = 1$ indicates a single stage. The latter option turns out to be more convenient when Ω_u serves as the base case of a general arbiter combining form, so we take

$$\Omega_u = \lambda((b, m), x) \cdot \langle (\lambda t \cdot \mathbf{Z}^t ((\mathbf{Z}^t \mathbf{R}(\mathbf{F}_t \langle D_1 b, D_0(m, x) \rangle \downarrow t, l^t)) \uparrow t)) \sum b b, \langle \mathbf{I}, \text{ARB} \rangle_{|b|-1} \rangle_{\delta_1^{|m|}} \quad (12.13)$$

as the definition of the mesh arbiter combining form hereafter.

12.2.2 Dendriform

A different way of designing an arbiter from a mesh is with a tree structure as shown in [Figure 12.3](#). The nodes in the tree effect arbitration by cooperating according to the following protocol.

- Every node arbitrates between two incoming requests from the environment or from its descendants (shown to the left in the figure), forwards a request to its ancestor on behalf of the winner, and waits for a grant from its ancestor.
- Upon receiving a grant from the ancestor, it relays the grant back to the winning descendant. (The root node in this design issues a grant to itself.)

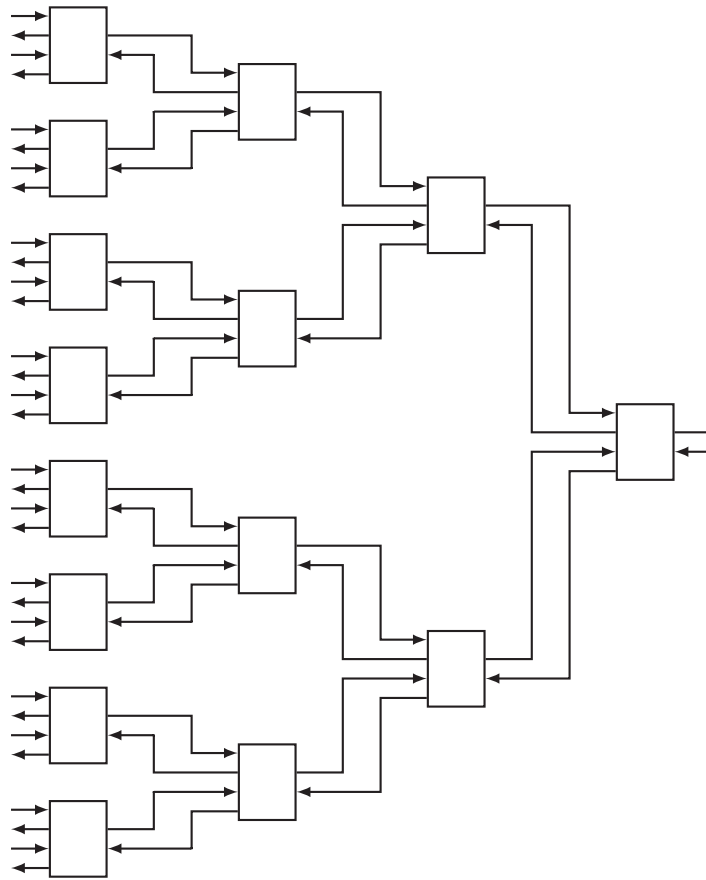


Figure 12.3: A branching arrangement of the blocks in [Figure 12.4](#) achieves arbitration with logarithmic critical paths.

- The winning client can assume mutually exclusive access to a resource protected by the arbiter from this point until the client issues a release signal.
- When the arbiter tree node receives a release signal from the winner, it forwards the release to its ancestor and waits for an acknowledgment.
- Upon receiving one, it relays the acknowledgment to the descendant that signaled the release, and may concurrently issue a pending request to its ancestor on behalf of the other descendant if any.

The sales pitch for this design is that a balanced tree requires no more than logarithmically many arbiters in any critical path, but takes no more than linearly many components of any type to build.

This asymptotic efficiency comes at the cost of some constant amount of overhead per input typically in excess of what would be required for a mesh. Each node in the tree is not just an arbiter, but an arbiter with additional state-holding circuitry including a decision wait as illustrated in

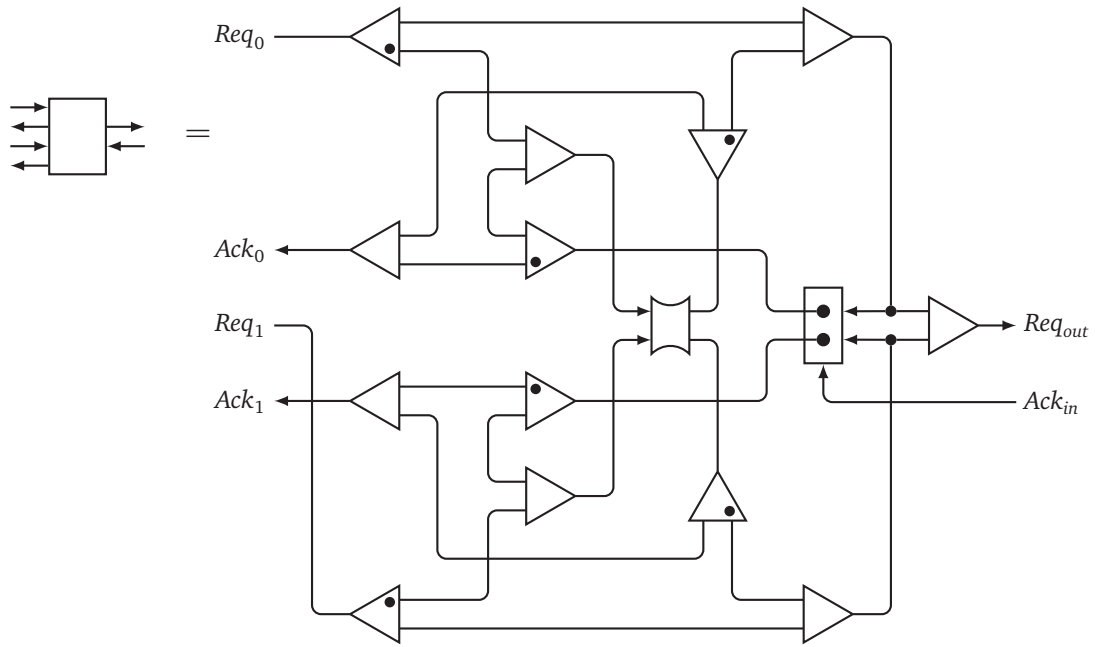


Figure 12.4: A dendriform arbiter cell $(D_3 D_2) \langle l, s, a \rangle$ mediates between s passive 4Φ ports on the left and one active 4Φ port on the right with an s -way arbiter a and an s -by-1 decision wait l , shown here for $s = 2$.

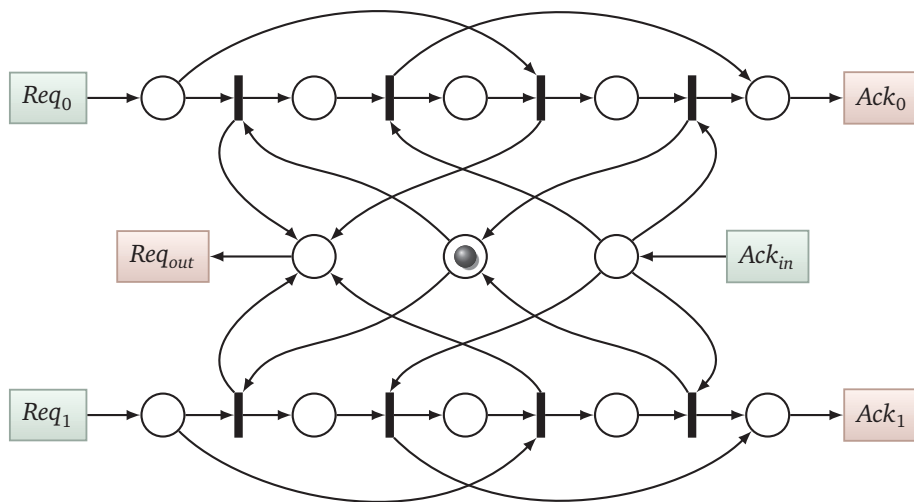


Figure 12.5: Petri net model of the circuit in Figure 12.4

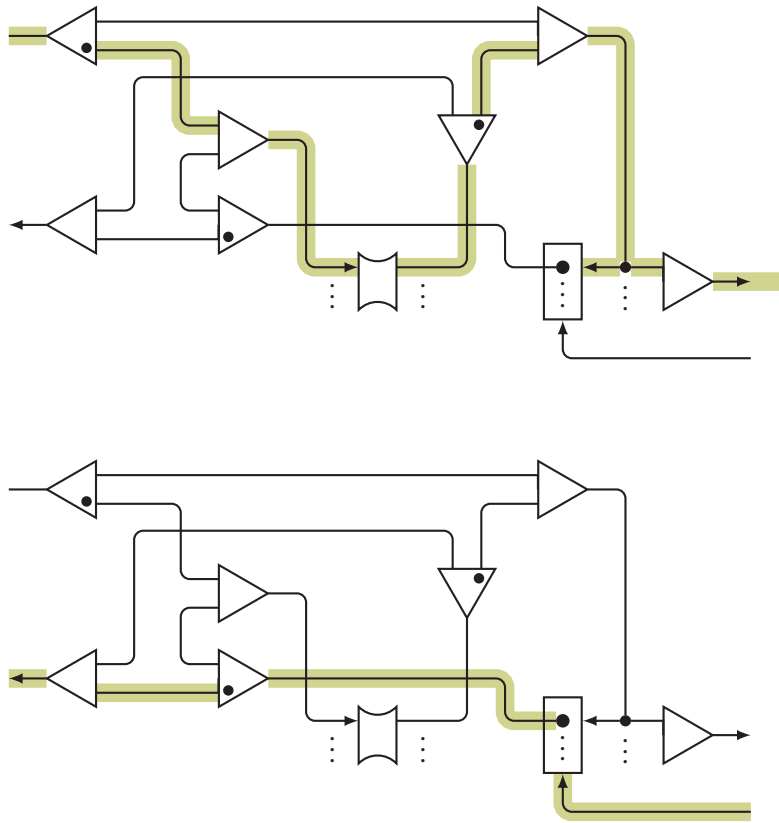


Figure 12.6: request phase (top) and grant phase (bottom) in one slice of an arbiter tree node

Figure 12.4 to implement the protocol described informally above and modeled by the Petri net in Figure 12.5. As for why the circuit needs to be this complicated, see Figure 12.6 and Figure 12.7 for a worm-level walk through a complete 4Φ cycle.

An obvious optimization on space would be to replace the self-acknowledging root node in Figure 12.3 with a single ARB primitive, or even the top two levels with a 4-way mesh. This optimization provides just one example of the potential benefits of mixing more than one form of arbiter in the same design, a matter we investigate in due course after considering each one in isolation.

However, constructing the dendriform arbiter with a view to mixing it in practice with other forms demands advance consideration along different lines. The two-port arbiter tree node shown in Figure 12.4 is straightforward to generalize to s ports for $s > 2$ with an s -by-1 decision wait, an s -way arbiter, and roughly one half of the rest of Figure 12.4 for each port. Taking $s = 3$ would allow ternary trees, for example, which would have fewer levels and conceivably shorter critical paths than binary trees with similar numbers of leaves. The internal s -way arbiter could be a mesh, a token ring (Section 12.2.3) or even another tree. The arbiter tree node calls for being defined in general enough terms to accommodate these possibilities.

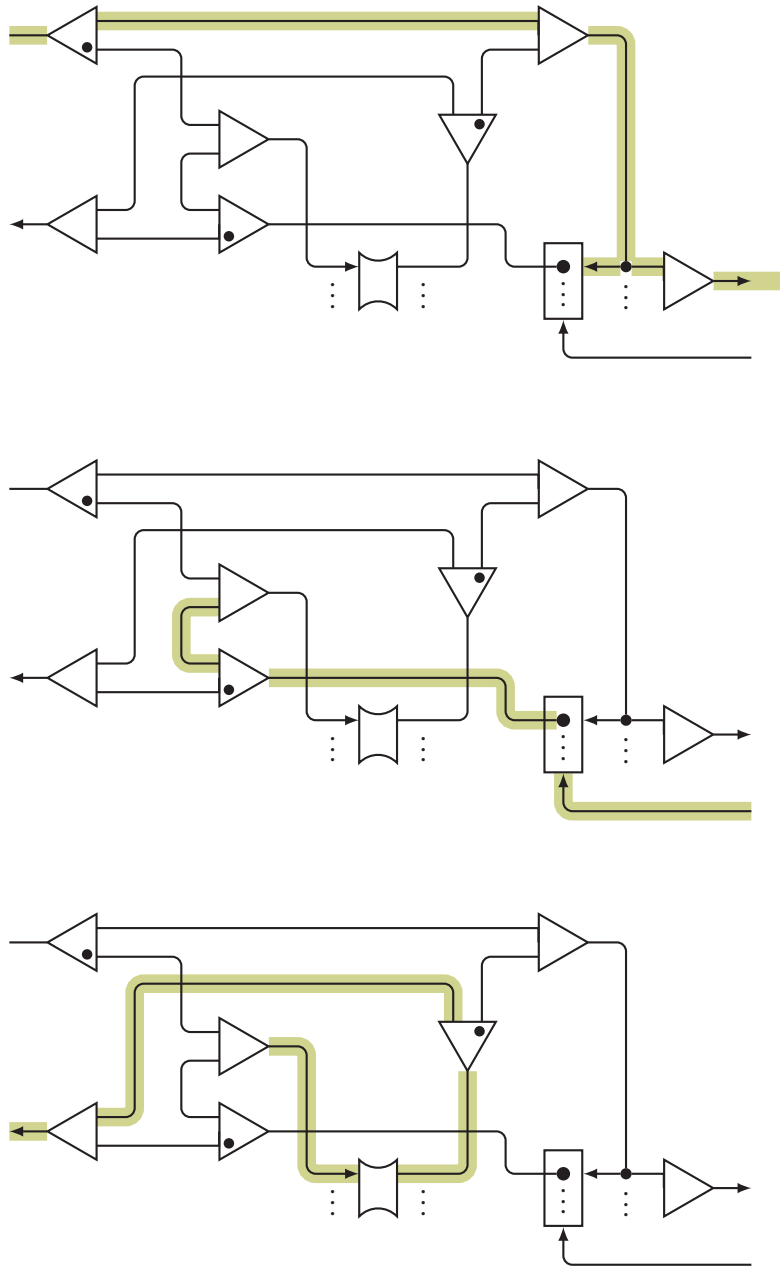


Figure 12.7: release phase (top) and acknowledge phase (center and bottom) in an arbiter tree node slice

Combining form

Rather than committing globally to a particular tree shape, we envision building a dendriform arbiter incrementally in levels having possibly varying attributes by a function

$$\Omega_{\vec{i}} : \mathbb{S}^* \times \mathbb{H}^* \times \mathbb{H} \rightarrow \mathbb{H}$$

taking a tuple $(c, x, y) \in \mathbb{S}^* \times \mathbb{H}^* \times \mathbb{H}$ to a dendriform arbiter $\Omega_{\vec{i}}(c, x, y) \in \mathbb{H}$ with just a single level of $|c| = |x|$ leaves below a root $y \in \mathbb{H}$, which can be an arbiter of any form including a tree. For each value of $0 \leq i < |x|$, the term $a = x_i \in \mathbb{H}$ is the arbiter inside the i -th leaf node of the arbiter to be built, and the corresponding term $c_i \in \mathbb{S}$ is the decomposition as defined in [Section 10.6.1](#) for the columnar decision wait

$$l = \hat{\Omega}_q c_i \in \mathbb{H} \quad (12.14)$$

appearing in the same node according to [Equation 10.30](#). The arity of the arbiter $a = x_i$ can be inferred from

$$s = ((\psi \Delta_q) c_i)_0 \quad (12.15)$$

by [Equation 10.26](#) and [Equation 10.27](#), being equal to the number of rows in the decision wait. The input arity of y is simply the number $|c|$ of leaves.

To start by constructing a typical arbiter tree node, [Figure 12.7](#) suggests that each of s “slices” has three TOGGLE primitives and three MERGE primitives, with each of the former connected to exactly two of the latter, as in

$$\mathbf{C}_6 \langle \text{TOGGLE}^3, \text{MERGE}^3 \uparrow 1 \rangle^s$$

or with all inputs and outputs both grouped into three buses of s lines each in $D_2 s$ according to

$$D_2 = \lambda s. \iota_{3s} \times s \times \mathbf{C}_6 \langle \text{TOGGLE}^3, \text{MERGE}^3 \uparrow 1 \rangle^s \times \iota_{3s} \times s \uparrow 1$$

including a roll of the inputs up by s for reasons to become clear shortly. This block is suitable for connecting the first s outputs to the s inputs on the arbiter a , the next s outputs to an array of s FORK primitives leading to the decision wait l and the output network MERGE s in

$$\mathbf{L}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(l \downarrow 1, \text{MERGE } s) \rangle$$

and the last s outputs to the s external acknowledgment lines, as in

$$\mathbf{D}_{2s} \langle D_2 s, \mathbf{R}(a, \mathbf{L}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(l \downarrow 1, \text{MERGE } s) \rangle) \downarrow 1 \rangle.$$

Outputs connected in this way constrain the first s inputs of $D_2 s$ to be connected to the outputs from the arbiter a , the next s inputs to be connected to the outputs from the decision wait l , and the rest to serve as the external request lines for consistency with [Figure 12.4](#), so we route the $2s$ lines through a bus \mathbf{I}^{2s} in

$$\mathbf{Z}^{2s} ((\mathbf{Z}^{2s} \mathbf{R}(D_{2s} \langle D_2 s, \mathbf{R}(a, \mathbf{L}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(l \downarrow 1, \text{MERGE } s) \rangle) \downarrow 1 \rangle), \mathbf{I}^{2s})) \downarrow 2s$$

which covers everything but the case of $s = 1$, an arbiter tree node having just one passive port. With no need for arbitration, requests and acknowledgments through a node with $s = 1$ can be relayed by a two-line bus \mathbf{I}^2 , so we write $(D_3 D_2) \langle l, s, a \rangle$ for the general arbiter tree node with D_3 defined as follows.

$$D_3 = \lambda d. \lambda \langle l, s, a \rangle. \langle \mathbf{Z}^{2s} ((\mathbf{Z}^{2s} \mathbf{R}(D_{2s} \langle d s, \mathbf{R}(a, \mathbf{L}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(l \downarrow 1, \text{MERGE } s) \rangle) \downarrow 1 \rangle), \mathbf{I}^{2s})) \downarrow 2s, \mathbf{I}^2 \rangle_{\delta_1}$$

This result features $s + 1$ inputs and $s + 1$ outputs, with the last of each pertaining to the active port interfacing the node with its ancestor in the tree.

To complete the construction, each leaf $h = (D_3 D_2) \langle l, s, a \rangle$ must be connected to the root y . The combination

$$\mathbf{Z}((\mathbf{ZR}(h \Downarrow 1, y)) \Downarrow 1)$$

of any h with y interfaces the active port on h with the last port on y , which is necessarily passive. We need only do the same with all terms in the list

$$(D_3 D_2)^* \langle \hat{\Omega}_q^* c, ((\psi \Delta_q)^* c)_0^\top, x \rangle^\top \in \mathbb{H}^*$$

of leaves h , which follows from [Equation 12.14](#) and [Equation 12.15](#), by folding over it as shown.

$$\Omega_{\hat{d}}(c, x, y) = (F_y \lambda(h, t). \mathbf{Z}((\mathbf{ZR}(h \Downarrow 1, t)) \Downarrow 1)) (D_3 D_2)^* \langle \hat{\Omega}_q^* c, ((\psi \Delta_q)^* c)_0^\top, x \rangle^\top \quad (12.16)$$

Decompositions

It would be useful for the purpose of global optimization to have a way of enumerating or sampling the space of dendriform arbiter designs similarly to the way [Equation 12.12](#) facilitates the enumeration of n -way mesh arbiters by $\nabla_u n$. A limited approach to this problem for the moment that we revisit more broadly in [Section 12.2.4](#) is simply to enumerate the population of lists $c \in \mathbb{S}^*$ of columnar decision wait decompositions that can be associated with the leaf nodes in dendriform arbiters of a given fixed arity. For an input arity n , there can be some number of leaves containing arbiters whose input arities sum to n , and therefore whose decision wait rows also sum to n . The rest is mostly a mathematical exercise.

The condition $\sum s = n$ for a list $s \in \mathbb{N}^*$ of row counts and a fixed $n \in \mathbb{N}$ mentioned above can be specified more constructively as $s \in B_7 n$ in terms of $B_7 : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}^*)$ defined by

$$B_7 = \lambda n. (F_{\emptyset} \lambda(h, t). \{\langle n - h \rangle\} \cup \bigcup_{l \in t} \wp(1 : l) \cup \bigcup_{j \in \mathcal{D}(l)} \wp(\lambda i. \delta_j^i + l_i)^* \iota_{|l|}) \iota_n \quad (12.17)$$

(among other alternatives) based on [Equation 12.2](#), which is to say that B_7 takes a natural number n to the set of all lists of positive natural numbers summing to n .

[Equation 12.17](#) is useful initially for enumerating decompositions of columnar decision waits of a given arity. Recall from [Chapter 10](#) that any r -by-1 decision wait decomposition $((p, d), \nu) \in \mathbb{S}$ features a list of dimensions $d = \langle s, \langle 1 \rangle \rangle$ satisfying $\sum s = r$ and a list of permutations

$$p \in \langle \iota_2 \rangle \parallel (\wp \iota_r)^1 \parallel \langle \iota_1 \rangle$$

meaning only the row permutation $p_1 \in \wp \iota_r$ can vary, with $p_0 = \iota_2$ fixing the columnar orientation and $p_2 = \iota_1$ being the only permutation of a single column. The list ν of subtrees can be empty for the cascading form, or can have $|\nu| = |s| = |d_0|$ for the quadrangular form, with each $\nu_i \in \mathbb{S}$ being the decomposition of a decision wait with s_i rows for $0 \leq i < |s|$. To avoid infinite regress, the quadrangular form is valid only for $|s| > 1$ and $s \neq \langle 1, 1 \rangle$. Hence we write $B_8 r \in \mathcal{P}(\mathbb{S})$ for the set of all r -by-1 decision wait decompositions with B_8 defined by this recurrence.

$$B_8(r) = \bigcup_{s \in B_7 r} ((\langle \iota_2 \rangle \parallel (\wp \iota_r)^1 \parallel \langle \iota_1 \rangle) \times \{\langle s, \langle 1 \rangle \rangle\}) \times (\lambda k. \langle (B_8^* s)^\top, \{\epsilon\} \rangle_k) \delta_1^{|s|} + \delta_s^{\langle 1, 1 \rangle} \quad (12.18)$$

We now make further use of [Equation 12.17](#) to enumerate dendriform arbiter decompositions as planned. A survey of dendriform arbiters could reasonably exclude any with fewer than three ports because either a single wire or an ARB primitive is adequate to implement them. It might also exclude those having only a single leaf, which are no improvement on the leaf's internal arbiter, and those whose every leaf has only a single passive port, which are equivalent to the root by itself. These latter exclusions have the added benefit of limiting the decompositions for any fixed arity to a finite set. Formally they amount to any dendriform arbiter with $n > 2$ ports worthy of consideration having $|s|$ leaves subject to the restriction

$$s \in (B_7 n) - (\mathbb{N}^1 \cup \{1\}^*)$$

such that the i -th leaf has s_i ports for all $0 \leq i < |s|$, and having a corresponding list

$$c \in (B_8^* s)^\top$$

of decision wait decompositions describing the decision waits in the leaves (that is, a list c in which each term c_i is a member of $B_8 s_i$). An expression for the set of lists of decision wait decompositions in terms of an arbitrary n

$$\bigcup_{s \in (B_7 n) - (\mathbb{N}^1 \cup \{1\}^*)} (B_8^* s)^\top$$

includes all such lists provided n exceeds two but is empty otherwise.

One way of summarizing this result for future reference might be in terms of a function taking an input arity $n \in \mathbb{N}$ to the set of lists c of decision wait decompositions, but if we opt instead for a function

$$\nabla_{\check{d}} : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{S}^{**})$$

taking the arity $n \in \mathbb{N}$ to the set of unit lists $\langle c \rangle$ of lists c of decision wait decompositions based on a definition

$$\nabla_{\check{d}} = \lambda n. \left(\bigcup_{s \in (B_7 n) - (\mathbb{N}^1 \cup \{1\}^*)} (B_8^* s)^\top \right)^1 \quad (12.19)$$

then it becomes more convenient subsequently to incorporate token ring arbiters within a common framework, whose description entails two decision wait decompositions per cell.

12.2.3 Token ring

Next in this tour of arbiter decompositions is the much anticipated token ring. A token ring arbiter involves a collection of cyclically connected cells depicted as squares in [Figure 12.8](#). Each cell interacts with the environment outside the ring using the same 4Φ protocol as a dendriform arbiter, and cooperates with its neighbors in such a way as to ensure that no more than one of them grants a request from the environment at any time.

- If a cell is authorized to grant a request from the environment, that cell and no other is said to be the holder of the token. The token is intangible but may be visualized as a soccer ball.
- If the environment makes a request to the cell holding the token, and that cell is not already serving another request, it grants the request immediately. Otherwise, it grants the request after finishing the previous one.

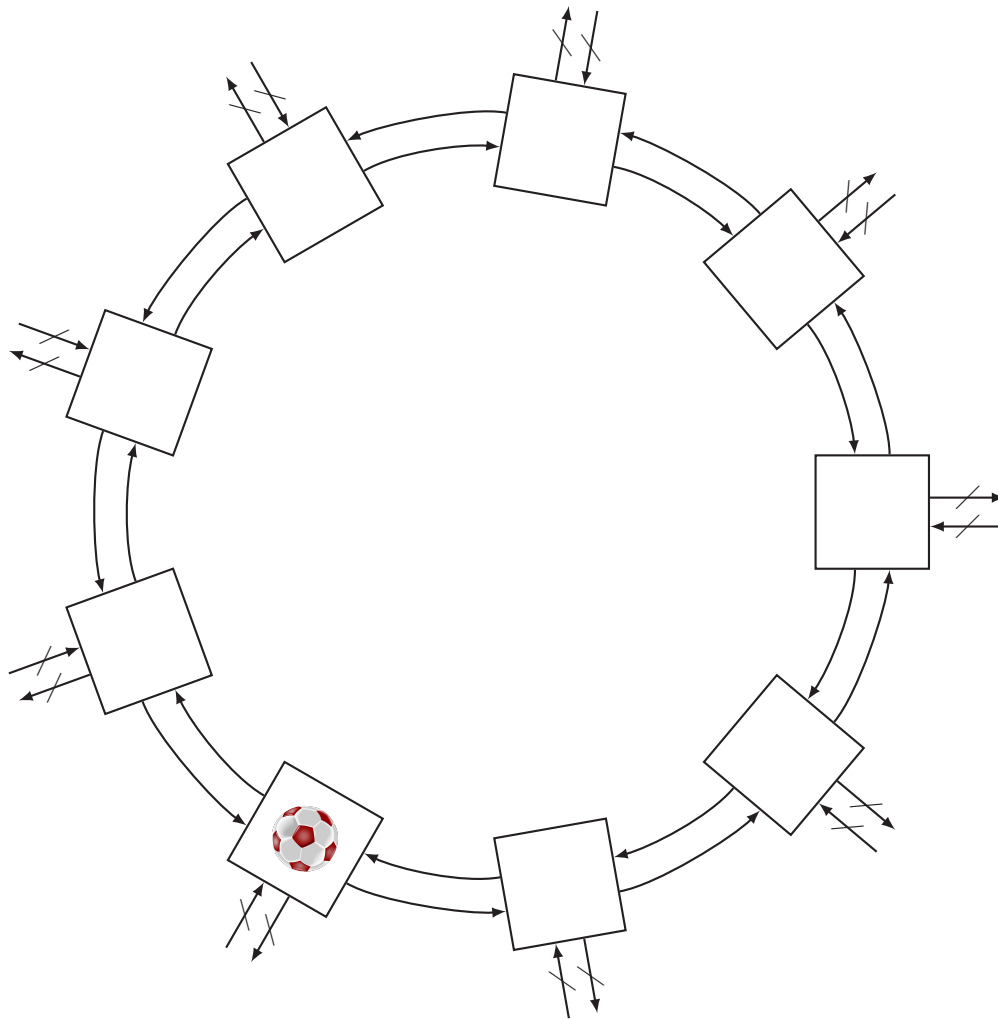


Figure 12.8: Cells in a token ring arbiter kick an imaginary token around the ring, and only the cell that holds the token can grant a request from the environment outside the ring.

- If the environment makes a request to a cell that is not holding the token, the cell requests the token from its counterclockwise neighbor and grants the request only after it subsequently receives the token from its counterclockwise neighbor. It then becomes the new holder of the token.
- If the cell holding the token receives a request for the token from its clockwise neighbor, and is not currently serving any request, it kicks the token to its clockwise neighbor and stops being the holder. If it is currently serving a request, it does so after finishing.
- If a cell that does not hold the token receives a request for the token from its clockwise neighbor, it requests the token from its counterclockwise neighbor. After receiving the token from its counterclockwise neighbor, it kicks the token immediately to its clockwise neighbor, holding it neither before nor afterwards.

The innards of a token ring arbiter cell depicted in [Figure 12.9](#) feature an inventory of one arbiter of any size and form, a columnar decision wait of a matching size to remember which external request is being served, a 2-by-2 decision wait to remember whether or not it holds the token, and miscellaneous glue, all of which are specified in detail presently, but the sales pitch is not so simple.



- One of the most important things about an arbiter supposedly is fairness. However, the token ring arbiter at least by some metrics is *unfair*. Given multiple concurrent requests, the cell holding the token has the biggest advantage, followed by the cells closest to it in the counterclockwise direction. Even worse, this advantage can persist over time if more requests to the same cells arrive fast enough.
- Another important thing about an arbiter unquestionably is performance. Whereas a request to a dendriform arbiter passes through only logarithmically many nodes if the tree is balanced, a request to a randomly selected cell in a token ring arbiter incurs a linear latency on average, even when uncontended, for the time needed by the token to propagate from the cell holding it to the cell granting the request.
- Cost is another criterion. The extra circuitry needed in a token ring arbiter cell in addition to the internal arbiter and columnar decision wait exceeds that of a dendriform arbiter tree node (designed under similar assumptions), not to mention a mesh.

In view of these ostensible drawbacks, why should anyone ever consider using a token ring arbiter? The answer follows from thinking like a computer architect. An insidiously pervasive phenomenon, **locality of reference** historically has mandated a storage hierarchy encompassing CPU caches, virtual memory, secondary storage, and network infrastructure for any competitively performant system to an extent that would be difficult to overstate [71]. Requests for a resource are overwhelmingly likely in many applications to follow repetitious patterns, not to be random in the vague sense implied above. Like a cache, a token ring arbiter “remembers” the most recently active cell and optimizes the next request to it and its neighbors. Token rings have been shown to outperform trees in applications characterized by long runs of accesses to nearby cells with relatively infrequent changes, such as neural networks [121], and the analysis starting in [Section 12.3](#) can take some of the guesswork out of this assessment. As for the high cost of a token ring compared to a tree, there are those who would take issue even with that [258].

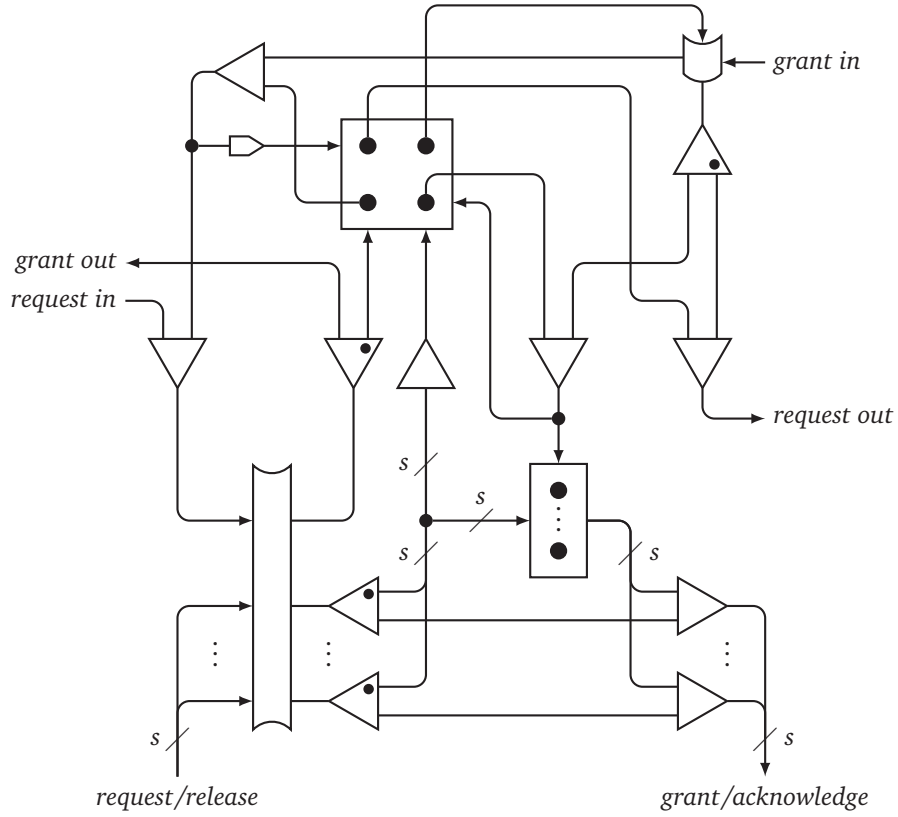


Figure 12.9: A token ring arbiter cell $D_7(s, a) D_6 \langle D_4(t, m), D_5(s, l) \rangle$ interacts with the environment below by s passive 4Φ ports, with a cell to the right by an active 2Φ port, and with a cell to the left by a passive 2Φ port, using an $s + 1$ -way arbiter a , an s -by-1 decision wait l , and a 2-by-2 decision wait m . The bottom row outputs of m are enabled only when the cell holds the token, shown here for the case $t = 0$ specifying an initially tokenless cell.

Combining form

A worm-level walk through Figure 12.9 analogous to Figure 12.6 and Figure 12.7 to understand the token ring arbiter cell implementation should pose no surprises and is left for an exercise as we proceed directly to the construction of its exact block or netlist representation (cf. Section E.2.3). To this end, we seek a combining form

$$\Omega_0 : \mathbb{S}^* \times \mathbb{S}^* \times \mathbb{H}^* \rightarrow \mathbb{H}$$

taking a triple $v = (c, b, x) \in \mathbb{S}^* \times \mathbb{S}^* \times \mathbb{H}^*$ to a token ring arbiter $\Omega_0 v \in \mathbb{H}$ of $|c| = |b| = |x|$ cells, with $c \in \mathbb{S}^*$ being a list of columnar decision wait decompositions having one term for each cell, $b \in \mathbb{S}^*$ being a list of one 2-by-2 decision wait decomposition for each cell, and $x \in \mathbb{H}^*$ being a list of arbiters also with one for each cell. The columnar decision waits need not have the same number

of rows in every cell, but any arbiter

$$a = x_i \in \mathbb{H}$$

for $0 \leq i < |x|$ is assumed to have exactly one more port than the number of rows

$$s = ((\psi \Delta_q) c_i)_0 \in \mathbb{N}$$

on the columnar decision wait

$$l = \hat{\Omega}_q c_i \in \mathbb{H}$$

in the same cell (cf. Equation 12.14 and Equation 12.15). The first port on the arbiter a mediates exchanges of the token with neighboring cells, and the other ports are accessible to the environment.

To have exactly one token in circulation requires one cell in the ring designed to hold the token initially and the rest not, so there are really two kinds of cells. However, the two designs are similar enough to be defined by a single expression parameterized in part by a value $t \in \{0, 1\}$ with $t = 1$ indicating the initial presence of a token. The difference amounts only to whether the 2-by-2 decision wait

$$m = \hat{\Omega}_q b_i \in \mathbb{H}$$

within the cell has a PUSH connected to its first row input, or its second. Following a convention whereby the bottom row outputs are enabled whenever the cell holds the token, a block

$$\mathbf{F}_2 \langle \mathbf{R}(\text{PUSH}, \mathbf{l}) \Downarrow t, m \rangle$$

behaves as a 2-by-2 decision wait with a signal initially sent to the bottom row only for $t = 1$, and to the top row otherwise.

Token storage The 2-by-2 decision wait and surrounding parts are primarily concerned with storage and querying of the token. As shown in Figure 12.9, each output from the 2-by-2 decision wait is connected to a separate MERGE except for the upper right, which is connected indirectly to all three by way of a SHUNT leading to one MERGE and a TOGGLE leading to the other two. Putting these five components into a block

$$\mathbf{F}_2 \langle \text{SHUNT}, \mathbf{Z}^2 \mathbf{R}(\text{TOGGLE}, \text{MERGE}^3 \leftarrow \gamma_2^1) \Downarrow 1 \rangle$$

achieves a solution $D_4(t, m) \in \mathbb{H}$ effecting the required connections with the decision wait after some amount of trial and error.

$$D_4 = \lambda(t, m). \mathbf{L}_4 \langle \mathbf{F}_2 \langle \mathbf{R}(\text{PUSH}, \mathbf{l}) \Downarrow t, m \rangle \uparrow 1, \mathbf{F}_2 \langle \text{SHUNT}, \mathbf{Z}^2 \mathbf{R}(\text{TOGGLE}, \text{MERGE}^3 \leftarrow \gamma_2^1) \Downarrow 1 \rangle \rangle$$

The result has five inputs and three outputs. The first input is the token-clearing control to be used when the cell relinquishes the token, the next is the token-setting control, the next two are the left and right column inputs to the decision wait used for querying the token, and the last is the grant input associated with the active 2Φ port shown on the right side of Figure 12.9. The last output is the request associated with the active 2Φ port, and the other two are each destined for a FORK to be included presently, the first involved with clearing the token and the second with setting it. In a lengthy construction like this one, making some rough notes as in Figure 12.10 can be helpful for keeping track of the details.

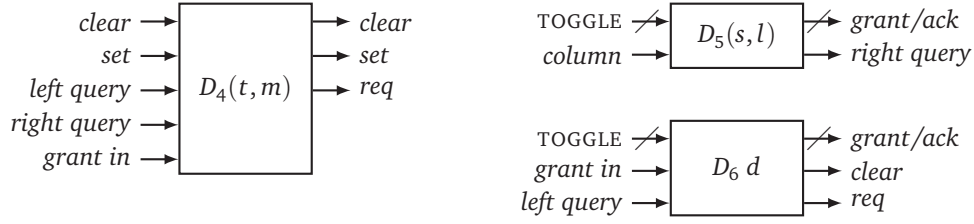


Figure 12.10: excerpt from a crib sheet purportedly having something to do with token ring arbiter cell building blocks

External port management While the 2-by-2 decision wait stores the token status for the cell, the columnar decision wait l keeps track of which 4Φ port has signaled a request, if any, and also withholds a grant if necessary until the token is acquired. As such, it needs s rows, one for each port, as well as a TOGGLE, a FORK, and a MERGE for each row, and an s -way MERGE network to query the token status on behalf of all of them. Proceeding outwards from a block

$$\mathbf{R}(\text{MERGE } s, l)$$

with s inputs to the MERGE followed by s row inputs to l enables $2s$ connections from a FORK network

$$\mathbf{F}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(\text{MERGE } s, l) \rangle$$

and another s connections from the s dotted TOGGLE outputs to the FORK inputs

$$\mathbf{F}_s \langle \text{TOGGLE}^s \uparrow_2^1, \mathbf{F}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(\text{MERGE } s, l) \rangle \rangle$$

which would leave the other s TOGGLE outputs at the beginning, followed by a single MERGE output, followed by the s decision wait outputs. Rolling the decision wait outputs to the beginning allows a block $D_5(s, l) \in \mathbb{H}$ with

$$D_5 = \lambda(s, l). \mathbf{F}_{2s} \langle \mathbf{F}_s \langle \text{TOGGLE}^s \uparrow_2^1, \mathbf{F}_{2s} \langle \text{FORK}^s \uparrow_2^1, \mathbf{R}(\text{MERGE } s, l) \rangle \rangle \downarrow s, \text{MERGE}^s \uparrow_2^1 \rangle$$

incorporating an output MERGE network to drive each of s external grant outputs from the corresponding pair of TOGGLE and decision wait outputs. The s -way MERGE output comes last on $D_5(s, l)$, and the column input to l is its last input.

Glue To combine these two parts of the cell, we first add the FORK pair with feedback paths to connect each of the two MERGE outputs on $D_4(t, m)$ to the appropriate respective token control inputs

$$\mathbf{Z}^2 \mathbf{R}(D_4(t, m), \text{FORK}^2 \uparrow_2^1)$$

and roll the grant input and the unused token clearing FORK output out of the way on a block

$$(\mathbf{Z}^2((\mathbf{Z}^2 \mathbf{R}(D_4(t, m), \text{FORK}^2 \uparrow_2^1))) \Downarrow 2$$

having the last column input on the 2-by-2 decision wait in the last input position and the unused token setting FORK output in the last output position. Then rolling the s -way MERGE output and the decision wait column input of $D_5(s, l)$ both to the top of

$$D_5(s, l) \Downarrow 1$$

enables the required connection from the s -way MERGE output to the last column of the 2-by-2 decision wait in

$$\mathbf{ZR}(D_5(s, l) \Downarrow 1, (\mathbf{Z}^2((\mathbf{Z}^2\mathbf{R}(D_4(t, m), \text{FORK}^2\Gamma_2^1)) \Downarrow 2)) \Downarrow 1)$$

along with the one from the token-setting FORK output to the columnar decision wait column input.

$$\mathbf{Z}((\mathbf{ZR}(D_5(s, l) \Downarrow 1, (\mathbf{Z}^2((\mathbf{Z}^2\mathbf{R}(D_4(t, m), \text{FORK}^2\Gamma_2^1)) \Downarrow 2)) \Downarrow 1) \Downarrow 1) \Downarrow 1)$$

The result has s inputs to the TOGGLE array due to $D_5(s, l)$ followed by the 2Φ grant input and the first column input to the 2-by-2 decision wait in that order. The s outputs from the MERGE array are followed by a token clearing FORK output and the 2Φ request output in that order. It may be more easily denoted as $D_6\langle D_4(t, m), D_5(s, l) \rangle$ with $D_6 : \mathbb{H}^2 \rightarrow \mathbb{H}$ defined by

$$D_6 = \lambda d. \mathbf{Z}((\mathbf{ZR}(d_1 \Downarrow 1, (\mathbf{Z}^2((\mathbf{Z}^2\mathbf{R}(d_0, \text{FORK}^2\Gamma_2^1)) \Downarrow 2)) \Downarrow 1) \Downarrow 1) \Downarrow 1).$$

Arbiter integration The rest of the cell consists of the arbiter a with a MERGE and a TOGGLE connected to its first port

$$\mathbf{F}\langle \text{MERGE}, a, \text{TOGGLE} \rangle$$

allowing the remaining s outputs from the arbiter to connect to the TOGGLE array exposed as the first s inputs by $d = D_6\langle D_4(t, m), D_5(s, l) \rangle$

$$\mathbf{F}_s\langle \mathbf{F}\langle \text{MERGE}, a, \text{TOGGLE} \rangle, d \rangle$$

and the first output from the TOGGLE to connect to the remaining column input on the 2-by-2 decision wait still exposed as the last input to d .

$$\mathbf{ZF}_s\langle \mathbf{F}\langle \text{MERGE}, a, \text{TOGGLE} \rangle, d \rangle$$

One further connection from the token clearing FORK in the penultimate output position on d to the MERGE leading to the arbiter is expressible by

$$\mathbf{Z}((\mathbf{ZF}_s\langle \mathbf{F}\langle \text{MERGE}, a, \text{TOGGLE} \rangle, d \rangle) \Downarrow 2)$$

which leaves a block having s inputs to the arbiter followed by the 2Φ grant input and then the 2Φ request input shown at the left of Figure 12.9 in that order. The outputs are the 2Φ request, the 2Φ grant, and the s MERGE outputs in that order. More useful for a cascade of cells would be to have the handshake lines in a matching order, the input bus last, and the output bus first on each cell, as in $D_7(s, a)$ d defined by

$$D_7 = \lambda(s, a). \lambda d. (\mathbf{Z}((\mathbf{ZF}_s\langle \mathbf{F}\langle \text{MERGE}, a, \text{TOGGLE} \rangle, d \rangle) \Downarrow 2)) \Downarrow 2 \times \langle 1, 0 \rangle.$$

Ring cascade With the work of constructing the cell complete, we turn to the matter of connecting multiple cells in a ring. Separate values of the parameters t , l , m , s , and a for each cell can be inferred from the given parameters c , b , and x by $D_8(c, b, x)$ according to

$$D_8 = \lambda(c, b, x). \langle 1 : (0 \stackrel{|x|}{\ll} 1), \hat{\Omega}_q^* c, \hat{\Omega}_q^* b, ((\Psi \Delta_q)^* c)_0^T, x \rangle^T \quad (12.20)$$

so that $D_8(c, b, x)_0$ is the list of parameters $\langle t, l, m, s, a \rangle$ pertaining to the first cell, $D_8(c, b, x)_1$ pertains to the next, and so on. The value $t = 1$ is associated only with the first cell to make it the only one that initially holds the token in a cascade

$$\mathbf{U}_2 (\lambda \langle t, l, m, s, a \rangle. D_7(s, a) D_6 \langle D_4(t, m), D_5(s, l) \rangle)^* D_8(c, b, x)$$

which lacks only the connections between the 2Φ handshake ports at either end to make a token ring arbiter. Hence we take the following as the definition overall.

$$\Omega_o(v) = \mathbf{Z}^2 (\mathbf{U}_2 (\lambda \langle t, l, m, s, a \rangle. D_7(s, a) D_6 \langle D_4(t, m), D_5(s, l) \rangle)^* D_8 v \times \langle 1, 0 \rangle \Downarrow 2) \quad (12.21)$$

Decompositions

A step toward sampling the space of token ring arbiters in the interest of optimization would be to enumerate the possible combinations of decision waits in their cells, of which there is one columnar and one 2-by-2 in each. Despite having fixed dimensions, 2-by-2 decision waits admit eight alternative decompositions

$$q = ((\wp \iota_2)^3 \times \{\langle 2, 2 \rangle\}) \times \{\epsilon\} \in \mathcal{P}(\mathbb{S}) \quad (12.22)$$

when all permutations and rotations are taken into account. Taking $r \in B_7 n - \mathbb{N}^1$ to be a non-unit list of positive natural numbers summing to an arity n by [Equation 12.17](#) gives rise to the set

$$(B_8^* r)^\top \in \mathcal{P}(\mathbb{S}^*)$$

of lists of decompositions of columnar decision waits whose row dimensions sum to n by [Equation 12.18](#). Each list $c \in (B_8^* r)^\top$ of columnar decision wait decompositions along with a matching list $b \in q^{|\text{cl}|}$ of 2-by-2 decision wait decompositions could then describe all of the decision waits in a particular token ring arbiter of n ports.

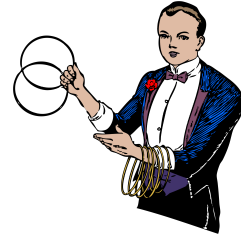
To make it official, let $\nabla_o n \in \mathcal{P}(\mathbb{S}^{*2})$ denote the set of lists $\langle c, b \rangle \in \mathbb{S}^{*2}$ where $c \in \mathbb{S}^*$ is any list of columnar decision wait decompositions whose rows sum to n and $b \in \mathbb{S}^{|\text{cl}|}$ is a list of the same number of 2-by-2 decision wait decompositions with ∇_o defined as follows.

$$\nabla_o = \lambda n. \bigcup_{r \in B_7 n - \mathbb{N}^1} \bigcup_{c \in (B_8^* r)^\top} (\mu \lambda b. \langle c, b \rangle) (((\wp \iota_2)^3 \times \{\langle 2, 2 \rangle\}) \times \{\epsilon\})^{|\text{cl}|} \quad (12.23)$$

The range of $r \in B_7 n - \mathbb{N}^1$ has the deliberate effect of excluding token ring arbiters with just one cell from this survey, both to keep the set $\nabla_o n$ finite for all $n \in \mathbb{N}$ and to exclude obviously suboptimal designs.

12.2.4 General

As noted on page 377, it may be advantageous to build an arbiter using more than one kind of decomposition. In addition to the example of using a mesh as the root of a tree, one might envision trees of rings, rings of trees, rings of rings nested to any depth, *etc.*, the last being analogous to a multi-level CPU cache in terms of the discussion on page 383. Performance advantages might also accrue from permuting the inputs on individual arbiters to fine-tune the critical path lengths within a larger construction, a



matter mostly ignored up to this point. Informed engineering decisions would depend on a way of automatically enumerating, searching, or sampling the space of arbiters built by any combination of decompositions, and more crucially a way of evaluating them.

Postponing the evaluation aspect for the moment, we can dispense with the easier part at least by capturing any arbiter decomposition as member of the set

$$\mathbb{A} = \mathcal{O}(\mathbb{N}^* \times ((\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}) \cup \mathbb{S}^{**}))$$

of ordered trees $((p, d), v)$, where $p \in \mathbb{N}^n$ is a permutation on the inputs of an n -way arbiter described by the tree, $v \in \mathbb{A}^*$ is a list of the decompositions of the arbiters that make it up, and d is either

- a pair $(b, m) \in \mathbb{N}^{*n} \times \mathcal{P}(\mathcal{R}(\iota_n))^{**}$ describing a mesh arbiter with n ports as proposed in [Section 12.2.1](#)
- a unit list $\langle c \rangle \in \mathbb{S}^{*1}$ of decompositions $c \in \mathbb{S}^*$ of the columnar decision waits in the leaves of a dendriform arbiter as proposed in [Section 12.2.2](#)
- or a two-item list $\langle c, b \rangle \in \mathbb{S}^{*2}$ of columnar decision wait decompositions $c \in \mathbb{S}^*$ and 2-by-2 decision wait decompositions $b \in \mathbb{S}^{|\mathbb{C}|}$ for a token ring as proposed in [Section 12.2.3](#).

Not all members of \mathbb{A} have sensible interpretations as arbiter decompositions unless the arities and permutation lengths are consistent throughout. However, for a fixed positive arity $n \in \mathbb{N}$, any member of the set $\nabla_a n \in \mathcal{P}(\mathbb{A})$ is a valid decomposition of an n -way arbiter by construction of a function

$$\nabla_a : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{A})$$

defined by the recurrence

$$\nabla_a = \lambda n. \bigcup_{(p,d) \in (\mathcal{P} \iota_n) \times ((\nabla_a n) \cup (\nabla_d n) \cup \nabla_o n)} \{(p, d)\} \times (\lambda v. (\nabla_a^* v)^\Gamma) \begin{cases} (\lambda c. c \parallel \langle |c| \rangle) ((\psi \Delta_q)^* d_0)_0^\Gamma & \text{if } d \in \mathbb{S}^{*1} \\ (\lambda c. ((\psi \Delta_q) c)_0 + 1)^* d_0 & \text{if } d \in \mathbb{S}^{*2} \\ (\lambda(b, m). (\lambda c. |c|)^* \flat \langle m, \epsilon \rangle_{\delta_1^{|m|}}) d & \text{otherwise} \end{cases} \quad (12.24)$$

and by [Equation 12.12](#), [Equation 12.19](#) and [Equation 12.23](#), where each of the three cases specifies the arities assumed about the list x of arbiters parameterizing the combining form $\Omega_{\bar{d}}$, Ω_o or Ω_u relevant to the type of decomposition d . Because Ω_u is defined by [Equation 12.13](#) to infer either a wire or a primitive arbiter without regard for the parameter x when there is only one stage $|m| = 1$, the subtrees of the decomposition can be omitted in this case. The set of all valid decompositions in \mathbb{A} follows as the union of $\nabla_a n$ for $n \in \mathbb{N} - \{0\}$, and the validity of a given tree $((p, d), v) \in \mathbb{A}$ is equivalent to the condition

$$((p, d), v) \in \nabla_a |p|.$$

An arbiter decomposition $t \in \mathbb{A}$ enables considerable analysis (to follow presently) without need of the fully elaborated block or netlist form of the arbiter it describes, but omits none of the information needed for its construction. We can think of a decomposition $t \in \mathbb{A}$ as an abstract or intermediate representation of a circuit $\Omega_a t \in \mathbb{H}$ available on demand through a function

$$\Omega_a : \mathbb{A} \rightarrow \mathbb{H}$$

defined by the recurrence

$$\Omega_a = \Lambda \lambda((p, d), \nu). (\lambda a. p \times a \times p) \begin{cases} \Omega_i(d_0, \nu \uparrow |d_0|, \nu_{|d_0|}) & \text{if } d \in \mathbb{S}^{*1} \\ \Omega_o(d_0, d_1, \nu) & \text{if } d \in \mathbb{S}^{*2} \\ \Omega_u(d, \nu) & \text{otherwise} \end{cases} \quad (12.25)$$

and by Equation 12.13, Equation 12.16, and Equation 12.21. See Equation 10.1 for a reminder about the tree folding combinator notation.

With the transformation from a decomposition to a circuit fully determined by Equation 12.25, the job of designing arbiters reduces to one of choosing an **arbiter decomposition strategy**

$$\mathcal{U}_a : \mathbb{N} \rightarrow \mathbb{A}$$

from among those satisfying

$$\forall n \in \mathbb{N} - \{0\}. \mathcal{U}_a n \in \nabla_a n$$

to enable an arbiter generating function $\text{ARB} : \mathbb{N} \rightarrow \mathbb{H}$ by a definition

$$\text{ARB} = \Omega_a \circ \mathcal{U}_a \quad (12.26)$$

preferably better informed than Equation 9.23. The choice of a decomposition strategy is justifiable only with respect to a cost or performance metric, hence the preoccupation with metrics in the balance of this chapter.

12.3 Transfer functions

What makes one arbiter better than another? One way to find out is to send some requests to both of them and see what happens. A more thorough way is to calculate based on their decompositions what would happen in the event of any conceivable pattern of requests, but pursuing this course immediately poses an obvious problem: the response of an arbiter to a pattern of requests is unpredictable due to the element of non-deterministic choice involved.



Not to worry, we can settle for an answer in terms of probability distributions, but then there is still a problem. The grant probability distribution of a token ring arbiter would have to depend on which cell holds the token, and every time the arbiter grants a request, the token circulates to the cell granting it, thereby generally changing the distribution. As noted previously, biasing the response possibly for the better is the whole reason for using token rings, so ignoring this effect would make the forthcoming analysis useless for comparisons among the various decompositions. Moreover, to the extent that the choice of a grant is uncertain, the subsequent location of the token is also uncertain, so we are limited to drawing inferences based on a distribution of the token location rather than a definite value.

With grants and token locations causally influencing each other, and no better than probabilistic information about either, subtlety and small steps are the way forward.

- Seeking a behavioral description of the arbiter as a **transfer function** taking a set of concurrent requests in some form to the induced grant probabilities, we ask as a first step only for an **incremental transfer function** assigning to each possible grant its probability of being the first. It is reasonable to treat token distributions as temporarily fixed for this purpose.

- From the incremental transfer function, we then infer an **incremental token distribution** modeling the effect of the first grant on the rearrangement of tokens throughout all affected parts of the arbiter.
- Following the assumed release and acknowledgment of the first request, the incremental transfer function yields the probability of each remaining grant being the next, contingent on an updated token distribution and the resolution of prior requests.
- A calculation that amounts to performing these steps alternately leads to a **cumulative transfer function** assigning to each possible grant its probability being the *last*. The analogous **cumulative token distribution** evolves similarly.
- When all members of a set of concurrent requests are acknowledged and released, the next step is to calculate the effect of another set of concurrent requests sequentially following the first set using the previously obtained cumulative token distribution as a starting point. In doing so repeatedly we get a broad overview of the arbiter's time evolution.

Despite not saying much with any certainty, this analysis can lead to meaningful comparative cost and performance metrics in a statistical sense to be developed subsequently.

12.3.1 Probability vectors and distributions

Carrying out the plan outlined above requires establishing a few conventions about the description of requests, grants, and token distributions ahead of the main derivations.

Request probability vectors

For a set of concurrent requests to an arbiter with n ports, a list $r \in [0, 1]^n$ with $r_i = 1$ indicates a request to the i -th port for any $0 \leq i < n$, and with $r_i = 0$, it indicates the absence of such a request. This representation also admits values of $0 < r_i < 1$, which express the probability of a request whose status is not known with certainty. A list r in this context is called a **request probability vector** hereafter, and is notably not a distribution because its terms need not sum to unity. However, we assume independence so that inferences about joint request probabilities follow accordingly, at least until seeing what can be done about this restriction in [Section 12.4](#).

Grant probability vectors

Grants are describable similarly by a **grant probability vector** $g \in [0, 1]^n$ for an arbiter with n ports. There can be no more than one grant at a time from an arbiter, but a grant probability vector can have multiple non-zero terms when the specific port issuing the grant is not known with certainty. A single request probability vector describing multiple concurrent requests determines a sequence of grant probability vectors, with one for the first grant chronologically, one for the next after the first is released, and so on.

Probability vector examples

An example of the grant probability vector describing the first grant issuing from a primitive arbiter with a request probability vector $r \in [0, 1]^2$ is sketched in [Figure 12.11](#). Many seemingly complicated questions about probability yield without a struggle to a picture like this one. The arbiter at the

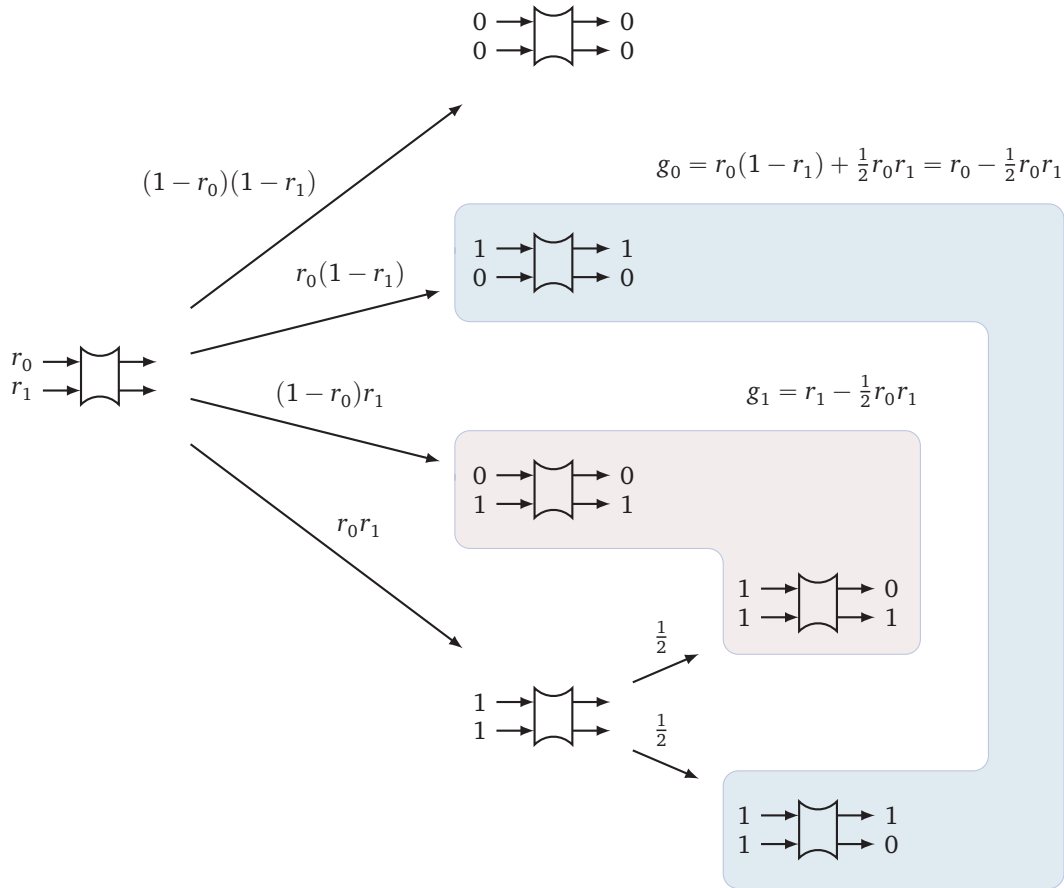


Figure 12.11: Independent requests made to a primitive arbiter with probabilities r_0 and r_1 imply initial grants with probabilities $r_0 - \frac{1}{2}r_0r_1$ and $r_1 - \frac{1}{2}r_0r_1$ as shown by this state diagram.

left can be viewed informally as an initial state with request probabilities r_0 and r_1 leading to four possible next states each characterized by requests known with certainty. Each arrow is labeled by the probability of the arbiter going that way, obtained as a product of request probabilities or complements thereof. The state reached with probability r_0r_1 due to both requests being present can result in either grant with probability $\frac{1}{2}$, as depicted by two further arrows.

As noted on page 364, probabilities of mutually exclusive events combine additively and probabilities of independent events combine multiplicatively. For present purposes, these rules of thumb mean the total probability of each grant is the sum of the products of the probabilities labeling the arrows leading to each final state where it appears. Simplifying a bit, we have

$$g = \langle r_0 - \frac{1}{2}r_0r_1, r_1 - \frac{1}{2}r_0r_1 \rangle \in [0, 1]^2 \tag{12.27}$$

as the grant probability vector. As this example shows, not all values of $r \in [0, 1]^*$ imply $\sum g = 1$, so neither is a grant probability vector generally a distribution, but unlike a request probability vector it can not sum to more than one.

Initial token distributions

Continuing in a similar vein suggests that the token distribution pertaining to an arbiter decomposition $t = ((p, d), v) \in \mathbb{A}$ could be specified by a list $k \in [0, 1]^{|v|}$ for t representing a token ring arbiter as indicated by $d \in \mathbb{S}^{*2}$. However, this convention would be inadequate for more complex decompositions such as nested rings. To provide a separate distribution for each token ring arbiter indicated at any level within the decomposition, the most workable solution entails a universe of **annotated decompositions**

$$\dot{\mathbb{A}} = \mathcal{O}(\mathbb{N}^* \times ((\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}) \cup \mathbb{S}^{**}) \times [0, 1]^*)$$

such that each tree $t' = ((p, d, k), v) \in \dot{\mathbb{A}}$ corresponds to a decomposition $t \in \mathbb{A}$ with the additional feature of a token distribution $k \in [0, 1]^*$ in each node. The correspondence is made explicit by a function

$$H_0 : \mathbb{A} \rightarrow \dot{\mathbb{A}}$$

taking any arbiter decomposition $t \in \mathbb{A}$ to an annotated decomposition $H_0 t \in \dot{\mathbb{A}}$ according to this recurrence.

$$H_0 = \Lambda \lambda((p, d), v). \begin{cases} ((p, d, 1 : 0^{\lfloor |p|-1}), v) & \text{if } d \in \mathbb{S}^{*2} \\ ((p, d, (1/|p|)^{\lfloor |p|}), v) & \text{otherwise} \end{cases} \quad (12.28)$$

Something to note about this definition is that it specifies only the **initial token distribution** for a token ring arbiter, which is always of the form $\langle 1, 0, 0, \dots \rangle$ because only the first cell can hold the token initially based on Equation 12.20. Another thing to note is that because tokens are not relevant to meshes or trees, simple uniform distributions are associated with them and are mostly ignored in the calculations to follow.

12.3.2 Incremental transfer function

The example shown in Figure 12.11 determines an incremental transfer function because it associates a grant probability vector g describing the first grant with any request probability vector r . Whereas the example pertains only to a primitive arbiter, in this section we seek the incremental transfer function for an arbiter specified by any given decomposition $t \in \dot{\mathbb{A}}$ in terms of a second order function

$$\dot{H}_1 : \dot{\mathbb{A}} \rightarrow ([0, 1]^* \rightarrow [0, 1]^*)$$

taking the decomposition to the transfer function $\dot{H}_1 t : [0, 1]^* \rightarrow [0, 1]^*$. Inevitably \dot{H}_1 is defined by a recurrence over $\dot{\mathbb{A}}$ based on a list $v \in ([0, 1]^* \rightarrow [0, 1]^*)^*$ of transfer functions already known hypothetically for the subtrees of the decomposition, and the root (p, d, k) . Derivations specific to meshes, trees, and token rings follow individually in preparation for the recurrence in general.

Mesh

Most of the difficulty in deriving the incremental transfer function for a mesh arbiter is due to most of the arbiters in it receiving their input signals from arbiters in previous stages, so it is worthwhile to tackle this aspect first. In a mesh arbiter with a broadcast network specified by a list $b \in \mathbb{N}^{**}$ as explained on page 370, each list $z \in \mathcal{R}(b)$ of broadcast zone sizes for an individual input signal s determines a list

$$(\lambda t. \iota_{z_t}^{\sum(z \uparrow t)})^* \iota_{|z|} \in \mathbb{N}^{**}$$

of the stage numbers in the mesh from front to back grouped by the broadcast zones of the signal numbered s . Except for the first zone, any arbiter in the i -th stage receiving that signal receives it only by way of arbiters in the last zone preceding the i -th stage, which would be those whose stages are numbered among the last term of the list

$$((\lambda t. \iota_{z_t}^{\sum(z \uparrow t)})^* \iota_{|z|}) \uparrow \mathcal{R}(t_i)^*$$

as given by $(h_0 b)_s i$ for $h_0 : \mathbb{N}^{**} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}^*)^*$ defined by

$$h_0 = (\lambda z. \lambda i. (\lambda l. \langle l_{|l|-1}, \epsilon \rangle_{\delta_f^\epsilon}) ((\lambda t. \iota_{z_t}^{\sum(z \uparrow t)})^* \iota_{|z|}) \uparrow \mathcal{R}(t_i)^*)^*. \quad (12.29)$$

Knowing the source of each signal is useful only if its request probability is also known. Whereas the request probabilities to arbiters in the first zone are assumed to be given, those of subsequent zones must be inferred from the grant probabilities of arbiters connected to them in the intervening zones. Although it is obviously not known in advance, we can build a list $g \in [0, 1]^{***}$ of lists of grant probability vectors incrementally by stages to meet this need, with $g_{ij} \in [0, 1]^*$ being that of the j -th arbiter in the i -th stage. For a mesh specified by a list $m \in \mathcal{P}(\mathbb{N})^{**}$ as explained in reference to Equation 12.9, a signal numbered s in the i -th stage would be routed to the j -th arbiter with

$$j = m_i^{-1} \left(\Psi \bigcup_{a \in \mathcal{R}(m_i)} a \times \{a\} \right) s$$

and if g is built to at least $i + 1$ stages, then the grant probability for the signal numbered s from the arbiter that grants it in the i -th stage (numbered from zero) follows as $((h_1 g)(m, s)) i$ for

$$h_1 : \mathbb{R}^{***} \rightarrow ((\mathcal{P}(\mathbb{N})^{**} \times \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{R}))$$

defined by

$$h_1 = \lambda g. \lambda(m, s). \lambda i. (\lambda j. (\lambda k. g_{ijk}) m_{ij}^\circ s m_i^{-1} \left(\Psi \bigcup_{a \in \mathcal{R}(m_i)} a \times \{a\} \right) s) \quad (12.30)$$

with $k = m_{ij}^\circ s$ identifying the port number associated with the signal numbered s on the j -th arbiter in the i -th stage.

Computing the i -th term to be appended to an extant list g of grant distributions requires assembling the request probability vectors locally applicable to all arbiters in the i -th stage and applying their respective transfer functions to them. For the j -th arbiter in the i -th stage, a request due to the signal numbered $s \in m_{ij}$ generally depends on the list of grant probabilities

$$((h_1 g)(m, s))^* (h_0 b)_s i \in [0, 1]^*$$

from the previous zone, and its probability is necessarily the product of these grant probabilities because the request does not reach the arbiter unless all of these grants are issued. In the special case of a stage within the first zone, the request probability is simply r_s , where r is the argument to the incremental transfer function under construction. In either case, the whole request probability vector relevant to the j -th arbiter in the i -th stage is obtained by

$$(\lambda s. (\lambda l. \langle \prod ((h_1 g)(m, s))^* l, r_s \rangle_{\delta_f^\epsilon}) (h_0 b)_s i)^* (m_{ij})^{\circ-1}$$

with s ranging over m_{ij} in ascending order, and hence the ensemble of request probability vectors for the i -th stage by $h_2 \langle h_0 b, h_1 g \rangle (m, r) i$ with h_2 defined as

$$h_2 = \lambda h. \lambda(m, r). \lambda i. (\lambda j. (\lambda s. (\lambda l. \langle \prod h_1(m, s)^* l, r_s \rangle_{\delta_f^\epsilon}) (h_0 b)_s i)^* (m_{ij})^{\circ-1})^* \iota_{|m_i|}. \quad (12.31)$$

The remaining task mentioned above about applying each arbiter's respective incremental transfer function to the corresponding request probability vector presupposes the list v of transfer functions available by hypothesis as noted previously. Then the grant probability vectors $g_0 \in [0, 1]**$ from the first stage would be

$$g_0 = (v \circ \iota_{|m_0|}) \triangle (h_2 \langle h_0 b, h_1 \epsilon \rangle (m, r)) \circ$$

by Equation 12.3, which immediately suggests a way of building the list of all $|m|$ lists of grant probability vectors as $h_3(b, m, v) r$ for h_3 defined by

$$h_3 = \lambda(b, m, v). \lambda r. (\lambda g. g \parallel \langle (v \circ \iota_{|m|}^{|b|g|}) \triangle (h_2 \langle h_0 b, h_1 g \rangle (m, r)) |g| \rangle)^{|m|} \epsilon. \quad (12.32)$$

The incremental transfer function overall would follow for the most part as

$$h_4((b, m), v) : [0, 1]^* \rightarrow [0, 1]^*$$

with

$$h_4 : ((\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}) \times ([0, 1]^* \rightarrow [0, 1]^*)^*) \rightarrow ([0, 1]^* \rightarrow [0, 1]^*)$$

defined as

$$h_4 = \lambda((b, m), v). (\lambda g. (b g_{|g|-1}) \circ (\overset{\circ}{b} m_{|m|-1})^{-1}) \circ h_3(b, m, v) \quad (12.33)$$

where the grant probability vectors of the back end stage represented by the last term of g are flattened into a single one and reordered by signal numbers, but let us hold that thought until deriving the analogous results for dendriform and token ring arbiters.

Dendriform

Incremental transfer functions for dendriform arbiters are more straightforward than for meshes or token rings. From a list $c = d_0 \in \mathbb{S}^*$ of columnar decision wait decompositions, we infer a list

$$s = ((\psi \Delta_q)^* c)_0^T \in \mathbb{N}^*$$

of input arities for the arbiters in the leaves, and by Equation 12.3, a list

$$g = v \triangle h_5(c, r) \in [0, 1]**$$

of grant probability vectors from the arbiters in the leaves based on transfer functions

$$v \in ([0, 1]^* \rightarrow [0, 1]^*)^{|c|}$$

and a function $h_5 : \mathbb{S}^* \times [0, 1]^* \rightarrow [0, 1]**$ defined by

$$h_5 = \lambda(c, r). (\lambda s. (\lambda i. r \circ \iota_{s_i}^{\sum(s_i i)})^* \iota_{|s|}) ((\psi \Delta_q)^* c)_0^T \quad (12.34)$$

that splits the request probability vector r into sublists whose lengths match the leaf arbiter arities. The rest of the derivation follows from a few basic observations.

- The j -th leaf makes a request to the root with probability $\sum g_j$, the sum of its grant probability vector, which makes the request probability vector to the root due to all of the leaves $\sum^* g$, resulting in a grant probability vector of $v_{|c|}(\sum^* g)$ from the root.

- Because each leaf waits for a grant from the root before forwarding a grant to the environment, a grant from the k -th port on the j -th leaf emerges only with probability $g_{jk}(v_{|c|}(\sum^* g))_j$, which is to say that it depends on both the leaf choosing the k -th port and the root choosing the j -th leaf as beneficiaries.
- The j -th leaf therefore accounts for a sublist $g_j \cdot (v_{|c|}(\sum^* g))_j$ of the overall grant probability vector $(h_6, h_5(c, r)) \nu$ with $h_6 : [0, 1]** \rightarrow (([0, 1]^* \rightarrow [0, 1]^*)^* \rightarrow [0, 1]^*)$ defined by

$$h_6 = \lambda h. \lambda v. (\lambda g. b (\lambda j. g_j \cdot (v_{|h|}(\sum^* g))_j^* \iota_{|g|}) (v \triangle h). \quad (12.35)$$

The resulting incremental transfer function analogous to $h_4(d, v)$ by Equation 12.33 for a dendriform arbiter follows as $\lambda r. (h_6, h_5(d_0, r)) \nu$.

Token ring

Attempting to express a token ring incremental transfer function similarly to the dendriform case leads immediately to the difficulty of deducing the correct request probability vectors. While the latter $|h_5(c, r)_j|$ ports on the arbiter in the j -th cell are exposed to the environment, the first port interfaces with the cell numbered $j - 1$ in a cyclic sense, so the request probability relevant to the first port is not known until the grant probability vector for the neighboring cell is known, an apparent impasse.

To dial down the difficulty temporarily, suppose quite unrealistically that the token is known to be held by the last or the highest numbered cell in the ring. Then it follows that the grant probability vector from the arbiter in the first cell is given by

$$g_0 = v_0(0 : h_5(c, r)_0) \in [0, 1]^*.$$

This expression features the first transfer function v_0 applied to a request probability vector specifying a request to the first port with probability zero and requests to the remaining ports with probabilities $h_5(c, r)_0$, the first sublist of the external request probability vector r . This value of g_0 is assured because the last cell, which already holds the token and is adjacent to the first, has no need to request the token.

On the other hand, any cell numbered j that does not hold the token requests it from the cell numbered $j + 1$ with probability $\sum g_j$ as determined by its incremental transfer function v_j , which is another way of saying it does so as surely as it grants any request at all. Continuing under this unrealistic assumption, we can build the whole list $g = h_7(v, h_5(c, r)) \in [0, 1]**$ of grant probability vectors according to

$$h_7 = \lambda(v, h). (\lambda g. g \parallel \langle v_{|g|}(\langle \sum g_{|g|-1}, 0 \rangle_{\delta_g} : h_{|g|}) \rangle)^{|v|} \epsilon \quad (12.36)$$

(cf. Equation 12.32).

In practice, the location of the token is not always the last cell, so this result needs to be generalized. If the token were in the first cell, then the grant probability vectors could be obtained by the same method with the ring and the requests rolled by one position

$$g \ll 1 \parallel g \parallel 1 = h_7(v \ll 1 \parallel v \parallel 1, h_5(c, r) \ll 1 \parallel h_5(c, r) \parallel 1)$$

and if the token were held by the l -th cell (numbered from zero), the result would be

$$g = (\lambda i. h_7(v \circ i, h_5(c, r) \circ i) \circ i^{-1}) (\iota_{|v|-l-1} \parallel \iota_{l+1}).$$

However, not even this result is general enough because no particular cell is known to hold the token. The token is only suspected of being in the l -th cell with probability k_l , where k is the token distribution given by the annotated decomposition whose incremental transfer function is presently sought. The right way forward is to evaluate the expectation of the grant probability vectors from the list

$$\dot{g} = ((\lambda l. k_l \cdot (\lambda i. h_7(v \circ i, h_5(c, v) \circ i) \circ i^{-1}) (t_{|v|-l-1}^{l+1} \parallel t_{l+1}))^* t_{|k|})^\top \in [0, 1]^{***}$$

wherein any term $\dot{g}_j \in [0, 1]^{**}$ is an ensemble of $|k|$ grant probability vectors associated with the j -th cell, with $\dot{g}_{jl} \in [0, 1]^*$ being the grant probability vector of the j -th cell when the token is held by the l -th cell. Then the expectation of $e = \dot{g}_j$ with respect to k follows as $\sum^* e^\top$, and the list of mean grant probability vectors as $h_8 \langle h_5(d_0, r), h_7 \rangle (k, v)$ according to

$$h_8 = \lambda h. \lambda(k, v). (\lambda e. \sum^* e^\top)^* ((\lambda l. k_l \cdot (\lambda i. h_1(v \circ i, h_0 \circ i) \circ i^{-1}) (t_{|v|-l-1}^{l+1} \parallel t_{l+1}))^* t_{|k|})^\top. \quad (12.37)$$

The externally observed grant probabilities are the cumulative concatenation of the individual grant probability vectors excluding the first term of each, so the actual result expressed as a function of the requests is

$$\lambda r. \flat (\lambda g. g \ll 1)^* h_8 \langle h_5(d_0, r), h_7 \rangle (k, v).$$

General

The two remaining requirements for an incremental transfer function defined by a recurrence over annotated decompositions $((p, d, k), v) \in \mathbb{A}$ are to define a result for the base case $v = \epsilon$, and to allow for the effects of the permutation p .

Base case The base case pertains to a mesh arbiter with only one stage, and either one or two ports, for which the list of subtrees in the decomposition is empty as noted previously in reference to [Equation 12.24](#). The arity of the arbiter can be inferred from the length $|p| \in \{1, 2\}$ of the permutation. A unit length permutation limits the arbiter to a single port implemented as a wire $\mathbb{1}$ by [Equation 12.13](#), and hence the identity function $\lambda r. r$ as its incremental transfer function. The other alternative is a primitive arbiter with an incremental transfer function

$$\lambda r. \langle r_0 - \frac{1}{2}r_0r_1, r_1 - \frac{1}{2}r_0r_1 \rangle$$

based on [Equation 12.27](#) and the discussion of [Figure 12.11](#).

Permutation networks The permutation p in the decomposition expresses an arbiter wrapped in input and output permutation networks described by p as in [Equation 12.25](#). When the environment requests with probability r_i to the i -th port visible on the outside of the combined arbiter and wrapper, the arbiter inside the wrapper sees a request on port p_i with probability r_i , and hence a request probability vector $r \circ p$ overall (cf. [Figure 8.15](#)). If the inner arbiter has an incremental transfer function f , then the combined system has $\lambda r. f(r \circ p)$ as its incremental transfer function.

From these last two points and [Equation 12.33](#) through [Equation 12.37](#), we have the following general form for the incremental transfer function of an arbiter described by any decomposition.

$$\dot{H}_1 = \Lambda((p, d, k), v). (\lambda f. \lambda r'. f(r' \circ p)) \begin{cases} \langle \lambda r. r, \lambda r. \langle r_0 - \frac{1}{2}r_0r_1, r_1 - \frac{1}{2}r_0r_1 \rangle \rangle_{|p|-1} & \text{if } v = \epsilon \\ \lambda r. (h_6 h_5(d_0, r)) v & \text{if } d \in \mathbb{S}^{*1} \\ \lambda r. \flat (\lambda g. g \ll 1)^* h_8 \langle h_5(d_0, r), h_7 \rangle (k, v) & \text{if } d \in \mathbb{S}^{*2} \\ h_4(d, v) & \text{otherwise} \end{cases}$$

12.3.3 Incremental token distribution

Now that $g = (\dot{H}_1 t) r$ predicts the first grant from an arbiter of decomposition $t = ((p, d, k), u) \in \dot{\mathbb{A}}$ and a request probability vector r , it would seem appropriate to follow through for the remaining requests, but this task is not just a matter of applying $\dot{H}_1 t$ to each of them. Each time a token ring arbiter grants a request, the token moves to the cell whose port grants it. This change affects how the arbiter handles subsequent requests, so it must be modeled by the token distribution k in the annotated decomposition t for correct results.

Grant gathering

If the cell in the token ring arbiter described by t that grants the request is known only up to a probability $w = \sum^* h_5(d_0, (\dot{H}_1 t) r)$, then the best estimate of the next token distribution is most probable where w is greatest,

$$k' = \frac{1}{\sum w} \cdot w$$

normalized to ensure $\sum k' = 1$ holds for the distribution. By Equation 12.34, this choice of k' assigns to the i -th cell a probability proportional to w_i , the probability of a grant issuing from anywhere in the i -th range of ports as determined by the summation $\sum h_5(d_0, (\dot{H}_1 t) r)_i$ of grant probabilities $g = (\dot{H}_1 t) r$ over the individual ports in that range. Alternatively, if t describes an arbiter of some other form, then the token distribution is not relevant, and if no grant is issued (due to no request being made), then the token distribution stays the same. To cover all cases, the root $(p, d, k') = \hat{G}(p, d, k) g$ of the adjusted decomposition can be defined as follows.

$$\hat{G} = \lambda(p, d, k) \cdot \lambda g \cdot \begin{cases} (\lambda w \cdot (p, d, (1/\sum w) \cdot w)) \sum^* h_5(d_0, g) & \text{if } d \in \mathbb{S}^{*2} \wedge g \notin \{0\}^* \\ (p, d, k) & \text{otherwise} \end{cases} \quad (12.38)$$

Request reckoning

Transforming t similarly at every level including the root is necessary if there are token rings nested within other forms, but more involved because the value of g parameterizing \hat{G} differs. Instead of being expressible as $g = (\dot{H}_1 t) r$ directly in terms of the request probability vector r overall, the token distribution in a node other than the root depends on whatever request probabilities are manifested locally to the arbiter it represents. Fortunately, this subproblem is already solved at least for the subtrees u immediately below the root in terms of the list $v = \dot{H}_1^* u \in ([0, 1]^* \rightarrow [0, 1]^*)^*$ of their incremental transfer functions if we temporarily fix the permutation p as the identity.

- If the root represents a dendriform arbiter, then there is a list $l = h_5(d_0, r)$ of request probability vectors to the leaves and an additional request probability vector $\sum^* v \triangle l$ to the root.
- If the root represents a mesh with $d = (b, m)$, then the list $h_3(b, m, v) r$ of grant probability vectors by Equation 12.32 determines a list $b \langle h_2 \langle h_0 b, h_1 h_3(b, m, v) r \rangle (m, r) \rangle^* \iota_{|m|}$ of request probability vectors by Equation 12.29 through Equation 12.31.
- If the root represents a token ring, then the first term g_0 of the i -th grant probability vector g in the list $h_8 \langle h_5(d_0, r), h_7 \rangle (k, v)$ by Equation 12.34, Equation 12.36 and Equation 12.37 is the first request probability to the arbiter in the succeeding cell, whose other request probabilities are the $(i + 1)$ -st term of $h_5(d_0, r)$ cyclically.

To summarize, let $\hat{R}(d, k, v) r \in [0, 1]**$ denote the list of request probability vectors that propagate to the subtrees of a decomposition with root (p, d, k) , where p is an identity permutation, v is the list of transfer functions of the subtrees, and r is the request probability vector imparted to the whole system, according to a function $\hat{R}(d, k, v) : [0, 1]^* \rightarrow [0, 1]**$ defined as follows.

$$\hat{R} = \lambda(d, k, v). \lambda r. \begin{cases} (\lambda l. l \parallel \langle \sum^* v \triangle l \rangle) h_5(d_0, r) & \text{if } d \in \mathbb{S}^{*1} \\ (\lambda i. b^* \langle \lambda g. \langle g_0 \rangle^* h_8 \langle i, h_7 \rangle (k, v) \circ (|i| - 1 : \iota_{|i|-1}, i)^\top \rangle) h_5(d_0, r) & \text{if } d \in \mathbb{S}^{*2} \\ (\lambda(b, m). b \langle h_2 \langle h_0 b, h_1 h_3(b, m, v) r \rangle (m, r) \rangle^* \iota_{|m|}) d & \text{otherwise} \end{cases}$$

Generalizing this result to unrestricted permutations is as simple as writing $\hat{R}(d, k, v) (r \circ p)$.

Token transit

Based on this preparation, the notion of updating the token distributions in a decomposition $t \in \hat{\mathbb{A}}$ can be captured formally by a second order function

$$\dot{H}_2 : \hat{\mathbb{A}} \rightarrow ([0, 1]^* \rightarrow \hat{\mathbb{A}})$$

parameterized by the given t , whereby $\dot{H}_2 t : [0, 1]^* \rightarrow \hat{\mathbb{A}}$ is the function that takes a request probability vector r to the annotated decomposition $(\dot{H}_2 t) r \in \hat{\mathbb{A}}$ differing from t only in its token distributions and only insofar as they might change due to a single grant prompted by request probabilities r . Awkward though it may be in prose, this style of specification enables a short definition of the function \dot{H}_2 as the solution to this recurrence.

$$\dot{H}_2(t) = (\lambda((p, d, k), u). \lambda r. (\hat{G}(p, d, k) (\dot{H}_1 t) r, (\dot{H}_2^* u) \triangle \hat{R}(d, k, \dot{H}_1^* u) (r \circ p))) t \quad (12.39)$$

12.3.4 Cumulative transfer function

Resuming now better equipped for the problem of modeling the effect of multiple concurrent requests leads in part to a cumulative transfer function

$$H_1 : \hat{\mathbb{A}} \rightarrow ([0, 1]^* \rightarrow [0, 1]^*)$$

of the same type as the incremental transfer function \dot{H}_1 but notably different in its construction and interpretation. Whereas the incremental transfer function parameterized by a decomposition t maps a request probability vector r to a grant probability vector $(\dot{H}_1 t) r$ pertaining to the first grant, the grant probability vector $(H_1 t) r$ pertains to the last grant after all others concurrently enabled by r are released and acknowledged. A corresponding cumulative token distribution function

$$H_2 : \hat{\mathbb{A}} \rightarrow ([0, 1]^* \rightarrow \hat{\mathbb{A}})$$

then follows with no further effort as

$$H_2(t) = (\lambda((p, d, k), u). \lambda r. (\hat{G}(p, d, k) (H_1 t) r, (H_2^* u) \triangle \hat{R}(d, k, H_1^* u) (r \circ p))) t \quad (12.40)$$

(cf. Equation 12.39) and fulfills the last prerequisite for passing from one set of concurrent requests to the next in the evaluation of any chosen cost or performance metric.

Deriving the grant probability vector for the last grant is mainly a math problem solvable without reference to the details of the arbiter under consideration. Lacking any obvious closed form or recursive solution, the problem is best broken into two parts.

- For the first part, we focus on computing the probability of a given sequence of n events encoded by $\langle s_0, s_1, \dots, s_{n-1} \rangle \in \wp \iota_n$ such that s_{n-1} is the port number of the first grant issued by an n -way arbiter with a given initial request probability vector r , s_{n-2} is port number of the next grant after the request to port s_{n-1} is released and acknowledged, and so on down to s_0 , the port number of the last grant. This calculation is feasible by adjusting the token distributions and request probability vectors accordingly at each step.
- For the second part, we identify the probability of any port s_0 being that of the last grant with the probability of some random sequence in $\wp \iota_n$ having s_0 as its first term. Following the rule of thumb for mutually exclusive events, this result is given by a summation of the probabilities obtained as proposed above over the set of such sequences. While we consider only the exact solution here, in practice this set may be large enough to mandate some form of sampling or extrapolation [35].

As discussed previously at great length, the probability of the first grant being on the i -th port of an arbiter with a decomposition t and a request probability vector r is $((\dot{H}_1 t) r)_i$, and as a side effect of these requests, t must be discarded in favor of a new decomposition $(\dot{H}_2 t) r$ for the next step toward evaluating the probability of a sequence. Along with modified token distributions, a modified request probability vector

$$(\lambda j \cdot (1 - \delta_j^i) r_j)^* \iota_{|r|}$$

derived from the original with the i -th request suppressed is appropriate for evaluating the joint probability of the first and second grants. If the latter were on the l -th port, it would be nearly correct to write the product

$$((\dot{H}_1 t) r)_i ((\dot{H}_1 (\dot{H}_2 t) r) (\lambda j \cdot (1 - \delta_j^i) r_j)^* \iota_{|r|})_l$$

as their joint probability were it not for a subtle issue: if the first grant has definitely occurred on the i -th port, as we are assuming temporarily it has, then the token distribution should say so unequivocally, but $(\dot{H}_2 t) r$ does not. The appropriate distribution for a token ring arbiter in this situation is zero-valued everywhere except the cell associated with the i -th port, where its value is unity. Conversely, not every token distribution in t is necessarily determined. If the i -th port belongs to a token ring in a leaf of a dendriform arbiter, the assumption of a grant from the i -th port implies nothing about the other leaves.

Rather than attempting *ad hoc* adjustments to all token distributions throughout the decomposition t , we can let the arbiter decide which distributions collapse and which survive by feeding it a second request probability vector

$$(\lambda j \cdot \delta_j^i)^* \iota_{|r|}$$

expressing a single request with certainty on the same port i as the assumed first grant. If a sublist of this request probability vector containing only zeros propagates anywhere within the arbiter, it leaves the associated token distribution unchanged by Equation 12.38, whereas any non-zero sublist leaves behind only a crisp clean binary token distribution indicative of certainty. Formally this effect is achieved by writing

$$((\dot{H}_1 t) r)_i ((\dot{H}_1 (\ddot{H}_2 t) (r, i)) (\lambda j \cdot (1 - \delta_j^i) r_j)^* \iota_{|r|})_l$$

in place of the expression above, with \ddot{H}_2 defined by

$$\ddot{H}_2 = \lambda t \cdot \lambda(r, i) \cdot (\dot{H}_2 (\dot{H}_2 t) r) (\lambda j \cdot \delta_j^i)^* \iota_{|r|}.$$

Generalizing this product from a pair of grants to a list is straightforward. A port number $i \in \mathcal{R}(s)$ in a permutation $s \in \mathbb{N}^*$, along with a partial product $q \in [0, 1]$, decomposition $t \in \nabla_a |s|$, and request probability vector $r \in [0, 1]^{|s|}$, determines the succeeding product $q((\dot{H}_1 t) r)_i$, decomposition $(\dot{H}_2 t) (r, i)$, and request probability vector $(\lambda j. (1 - \delta_j^i) r_j)^* \iota_{|r|}$ necessary for a function

$$\lambda(i, (q, (t, r))). (q((\dot{H}_1 t) r)_i, ((\dot{H}_2 t) (r, i), (\lambda j. (1 - \delta_j^i) r_j)^* \iota_{|r|}))$$

folded over s with any vacuous case result of the form $(1, u) \in [0, 1] \times ((\nabla_a |s|) \times [0, 1]^{|s|})$. The complete product follows as $(h_g u) s$ for $h_g : (\mathbb{A} \times [0, 1]^*) \rightarrow (\mathbb{N}^* \rightarrow [0, 1])$ given by

$$h_g = \lambda u. (\lambda(q, u'). q) \circ_{\mathcal{F}(1, u)} \lambda(i, (q, (t, r))). (q((\dot{H}_1 t) r)_i, ((\dot{H}_2 t) (r, i), (\lambda j. (1 - \delta_j^i) r_j)^* \iota_{|r|}))$$

and can be interpreted as the probability of the sequence of grants whose port numbers are listed in reverse by s .

The rest of the cumulative transfer function derivation follows first by building on this result to obtain the function

$$f = \Psi \Pi (\mu \lambda s. (s_0, h_g(t, r) s)) \wp \iota_{|r|} : \mathcal{D}(r) \rightarrow \mathcal{P}([0, 1])$$

induced by a given decomposition $t \in \mathbb{A}$ and request probability vector $r \in [0, 1]^*$ by [Equation 6.1](#), [Equation 6.7](#), and [Equation 5.1](#). For a given port number $s_0 \in \mathcal{D}(r)$ as an argument, $f s_0$ obtains the set of all probabilities $q = h_g(t, r) s \in [0, 1]$ of sequences of grants appearing on ports numbered in reverse by lists $s \in \wp \iota_{|r|}$ whose first term is s_0 . Then we can write $\sum_{q \in f i} q$ for the total probability of the i -th port being the last to exhibit a grant, and hence the following cumulative transfer function definition.

$$H_1 = \lambda t. \lambda r. (\lambda f. (\lambda i. \sum_{q \in f i} q)^* \iota_{|r|}) \Psi \Pi (\mu \lambda s. (s_0, h_g(t, r) s)) \wp \iota_{|r|}$$

12.4 Access patterns

Learning something useful about an arbiter from its transfer function depends on plugging useful arguments into it. For example, if the anticipated operating conditions were to imply some ports being busier than others, then testing the transfer functions of various designs on a request probability vector calibrated against empirical data to that effect would be an excellent way to find the one that performs best on average by optimizing the busier ports. However, this analysis by itself would neglect any spatial or temporal locality in the access pattern, whose effect could be more significant to performance overall, especially when token rings take advantage of it ([page 383](#)).

- **Spatial locality** is the tendency for a request to appear concurrently with another request regardless of their respective probabilities.
- **Temporal locality** is the tendency for a subsequent request to appear on the same port as a previous one more frequently than its probability alone would suggest.

Spatial and temporal locality can be modeled by conditional probabilities as defined in [Section 12.1.4](#), fortunately in a way that builds on the previous analysis without need of revision in [Section 12.4.1](#) and [Section 12.4.2](#) respectively, clearing the way for a precise concept of expectation of a metric with respect to a general access pattern in [Section 12.5](#).

12.4.1 Spatial locality

To start with a simple question, assuming a request probability vector $r \in [0, 1]^n$ in the sense proposed in [Section 12.3.1](#), what should we infer as the probability of a binary valued **request vector** $q \in \{0, 1\}^n$, where $q_i = 1$ holds if and only if there is *actually* a request on the i -th port? If the requests are spatially independent as assumed previously, then clearly the probability is the product

$$\prod (\lambda_i \cdot \langle 1 - r_i, r_i \rangle_{q_i})^* \iota_n$$

of the individual request probabilities or their complements, but if independence can not be assumed, then the probability could be anything within $[0, 1]$ subject only to the constraint that the probabilities of all request vectors $q \in \{0, 1\}^n$ sum to unity. Conversely, there should be a way to express any desired spatial locality consistently with the request probability vector r by letting some distribution $f : \{0, 1\}^n \rightarrow [0, 1]$ determine the probability $f q \in [0, 1]$ accordingly.

The space of all such distributions is larger than necessary to cover the access patterns of interest, for which a family of distributions

$$\Theta_n(r, c) : \{0, 1\}^n \rightarrow [0, 1]$$

parameterized by the request probabilities $r \in [0, 1]^n$ and the **spatial conditional probabilities** $c \in ([0, 1]^n)^n$ is adequate. The value of $c_{ij} \in [0, 1]$ specifies the conditional probability of a request on the i -th port given that there is concurrently a request on the j -th port. Aside from the obvious constraint

$$\forall i, j \in \mathcal{R}(\iota_n). i = j \Rightarrow c_{ij} = 1$$

we can dial in any values of $c_{ij} \in [0, 1]$ supported by empirical or theoretical studies to obtain $\Theta_n(r, c)$ as the solution to the system of 2^n linear equations in 2^n unknowns

$$\sum \Theta_n(r, c)^* (\lambda t \cdot \bar{t}_t) (\lambda v \cdot 1 + \delta_u^{u-\{v\}})^* \iota_n = \sum (\lambda i \cdot (r_i \prod_{j \in u} c_{ij}) / \sum r)^* \iota_n \quad (12.41)$$

for all $u \in \mathcal{P}(\mathcal{R}(\iota_n))$. (See [Equation 10.3](#) for a reminder about the notation \bar{t}_t .) It may give some consolation to note that this system although large is relatively sparse, with the number of non-zero coefficients growing only as 3^n , and that a correct solution is verifiable as such by these conditions.

$$r = (\lambda i \cdot \sum_{q \in \{0, 1\}^n} q_i (\Theta_n(r, c) q))^* \iota_n \quad c = (\lambda j \cdot \frac{1}{r_j} \sum_{q \in \{0, 1\}^n} q_i q_j (\Theta_n(r, c) q))^* \iota_n$$

12.4.2 Temporal locality

An access pattern parameterized by $r \in [0, 1]^n$ and $c \in ([0, 1]^n)^n$ exhibits temporal locality whenever the probability of a sequence $s \in (\{0, 1\}^n)^k$ of k consecutive request vectors relative to the sample space $(\{0, 1\}^n)^k$ differs from the probability

$$(\prod \Theta_n(r, c)^* s) / \sum_{t \in (\{0, 1\}^n)^k} \prod \Theta_n(r, c)^* t$$

it would normally have if there were no temporal locality (*i.e.*, the one fully determined by the product of the individual request vector probabilities). By favoring some sequences over others, a family of deliberately skewed distributions for arities n and sequence lengths k

$$\Theta_n^k(r, c, l) : (\{0, 1\}^n)^k \rightarrow [0, 1]$$

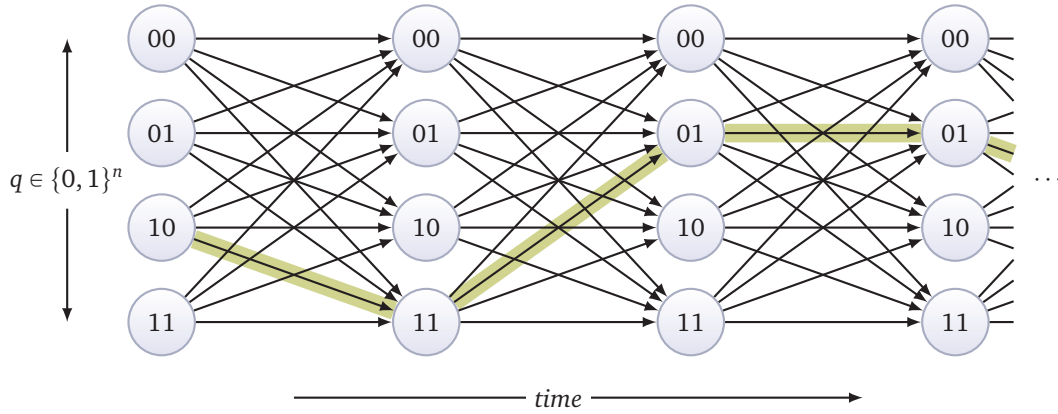


Figure 12.12: A k -partite graph associating a request vector with each node and a transition probability with each edge reveals the probability of a sequence $s = \langle \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 0, 1 \rangle, \langle 0, 1 \rangle \dots \rangle$ of k request vectors (among others) as the product of the initial request probability $\Theta_n(r, c) s_0$ and the transition probabilities from each request in the sequence to the next.

parameterized additionally by **temporal conditional probabilities** $l \in [0, 1]^n$ indicates a conditional probability l_i for a request on the i -th port given that the preceding request vector in the sequence also exhibits a request on the i -th port precisely when l satisfies

$$l = \left(\lambda w. \sum^* \left(\sum_{v \in w} \left(\lambda t. \frac{(\Theta_n^k(r, c, l) v^T) \sum (\lambda i. t_i t_{i+1})^* t_{k-1}}{(\sum_{b \in w} \Theta_n^k(r, c, l) b^T) \sum (t \mid k-1)} \right) v \right)^T \right) (\{0, 1\}^k - (\{0\}^* \parallel \{0, 1\}^1))^n$$

where each $v \in w$ is the transpose of a member of $(\{0, 1\}^n)^k$ excluding any terms $t \in \mathcal{R}(v)$ whose non-final terms $t \mid k-1$ are all zero. The partial result inferred from each instance of t in this expression is weighted by the probability of v^T given $v \in w$.

In practice r , c , and l are obtained empirically and a distribution $\Theta_n^k(r, c, l)$ consistent with this equation is sought for a value of k fixed by the available computing resources. To find it, we envision a directed k -partite graph as in Figure 12.12 and identify each sequence $s \in (\{0, 1\}^n)^k$ with a path.³ Associating a **transition probability** $\tilde{\Theta}_n(r, c, l) (q, u)$ with each edge in the graph reduces the problem to one of expressing $\Theta_n^k(r, c, l)$ as a product of an initial request probability with the transition probabilities along the subsequent path

$$\Theta_n^k(r, c, l) = \lambda s. (\Theta_n(r, c) s_0) \prod_{i=0}^{k-2} \tilde{\Theta}_n(r, c, l) (s_i, s_{i+1}) \tag{12.42}$$

where $\tilde{\Theta}_n(r, c, l) : \{0, 1\}^n \times \{0, 1\}^n \rightarrow [0, 1]$ applied to a pair (q, u) gives the conditional probability of the next request vector being u given that the current one is q , or intuitively the weight of an edge from node q to node u .

³This technique is popular in the financial derivatives literature, where the graph is often called a **lattice** [94], not to be confused with other usages of the term as in Appendix B.

Certain healthiness conditions constrain any contrivance of this ilk. Because $\vec{\Theta}_n(r, c, l)$ is a distribution, it satisfies

$$\sum_{u \in \{0,1\}^n} \vec{\Theta}_n(r, c, l)(q, u) = 1 \quad (12.43)$$

for all $q \in \{0, 1\}^n$, which can be visualized as the weights on the outgoing edges from each node summing to unity. Furthermore, the total probability $\Theta_n(r, c) u$ of every request vector $u \in \{0, 1\}^n$ must coincide with the sum

$$\Theta_n(r, c) u = \sum_{q \in \{0,1\}^n} (\Theta_n(r, c) q) (\vec{\Theta}_n(r, c, l)(q, u)) \quad (12.44)$$

of conditional probabilities contributed by all paths leading to it through nodes q in the previous partition class.

More specific to the problem at hand, we require

$$l_i = (\lambda v. \sum_{q \in v} \sum_{u \in v} \frac{\Theta_n(r, c) q}{\Theta_n(r, c) u} (\vec{\Theta}_n(r, c, l)(q, u))) \{t \in \{0, 1\}^n \mid t_i = 1 \wedge \Theta_n(r, c) t \neq 0\} \quad (12.45)$$

for $0 \leq i < n$ to enforce the given temporal conditional probabilities l . For example, with $n = 2$, the conditional probability l_0 of $u_0 = 1$ in a request u given $q_0 = 1$ in the preceding request q must be

$$\begin{aligned} l_0 &= \frac{(\Theta_n(r, c) \langle 1, 0 \rangle) (\vec{\Theta}_n(r, c, l) (\langle 1, 0 \rangle, \langle 1, 0 \rangle)) + (\Theta_n(r, c) \langle 1, 1 \rangle) (\vec{\Theta}_n(r, c, l) (\langle 1, 1 \rangle, \langle 1, 0 \rangle))}{\Theta_n(r, c) \langle 1, 0 \rangle} \\ &+ \frac{(\Theta_n(r, c) \langle 1, 0 \rangle) (\vec{\Theta}_n(r, c, l) (\langle 1, 0 \rangle, \langle 1, 1 \rangle)) + (\Theta_n(r, c) \langle 1, 1 \rangle) (\vec{\Theta}_n(r, c, l) (\langle 1, 1 \rangle, \langle 1, 1 \rangle))}{\Theta_n(r, c) \langle 1, 1 \rangle} \end{aligned}$$

if neither $\Theta_n(r, c) \langle 1, 0 \rangle$ nor $\Theta_n(r, c) \langle 1, 1 \rangle$ vanishes.

These conditions would establish a system of linear equations that could be solved for $\vec{\Theta}_n(r, c, l)$ at every point (q, u) were it not for the system being grossly underdetermined. With 4^n unknown transition probabilities and the constraints above implying only $2^{n+1} + n$ equations, we are free to choose the value of $\vec{\Theta}_n(r, c, l)$ at almost every point.

- A reasonable choice would have the conditional probability $\vec{\Theta}_n(r, c, l)(q, u)$ smoothly approach the total probability $\Theta_n(r, c) u$ as l approaches r , and match it exactly for $q \in \{0\}^n$ regardless of l .
- More generally, the conditional probability could be made to differ from the total probability by a factor of l_i/r_i for every i with $q_i = u_i = 1$, and by a factor of $(1 - l_i)/(1 - r_i)$ for every i with $q_i = 1$ and $u_i = 0$.

Following this plan, let $\vec{\Theta}_n(r, c, l)$ take values of

$$\vec{\Theta}_n(r, c, l)(q, u) = (\Theta_n(r, c) u) \left(\lambda f. \frac{fl}{\delta_0^{fr} + fr} \right) \lambda v. \prod (\lambda i. \langle 1, \langle 1 - v_i, v_i \rangle_{u_i} \rangle_{q_i})^* \iota_{|v|} \quad (12.46)$$

for all members (q, u) of a set $w \subset \{0, 1\}^n \times \{0, 1\}^n$ of $4^n - 2^{n+1} - n$ arbitrarily sampled points satisfying

$$\forall q \in \mathcal{D}(w). |w \cap (\{q\} \times \mathcal{R}(w))| < 2^n \wedge \forall u \in \mathcal{R}(w). |w \cap (\mathcal{D}(w) \times \{u\})| < 2^n$$

to avoid contradicting Equation 12.43 or Equation 12.44 and

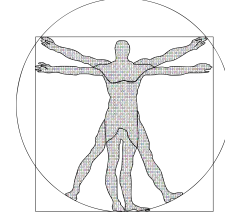
$$\forall i \in \mathcal{R}(t_n). \{0, 1\}^{i-1} \parallel \langle 1 \rangle \parallel \{0, 1\}^{n-i} \not\subseteq w$$

to avoid contradicting Equation 12.45, so that $\vec{\Theta}_n(r, c, l)$ induces a well defined distribution in Equation 12.42.

12.5 Metrics

Because there could be many quantitative questions about arbiters in need of good answers, and because not everyone is interested in the same ones, we refrain for a short while longer from getting more specific than necessary and refer only to a metric in the abstract. An arbiter metric for our purposes can be any freely chosen function

$$Q : \dot{\mathbb{A}} \rightarrow (\{0, 1\}^* \rightarrow \mathbb{R})$$



allowed to depend in any way at all on the annotated decomposition $t \in \dot{\mathbb{A}}$ of an arbiter (and implicitly, on its token distributions) to yield a function $Q t : \{0, 1\}^* \rightarrow \mathbb{R}$, which may optionally depend further on a current request vector $r \in \{0, 1\}^*$ to yield a value $(Q t) r \in \mathbb{R}$ quantifying some cost or performance characteristic of interest. While possibly useful by itself, a metric can often yield further insight when averaged over a variety of sequences of requests representative of actual operating conditions, and this calculation is expressible without any other assumptions about the metric in terms of the theory developed in Section 12.3 and Section 12.4. A precise account of this idea follows in Section 12.5.1, with some concluding remarks and implications in Section 12.5.2.

12.5.1 Expectation

A decomposition $t \in \mathbb{A}$ of an arbiter with the token distribution it has when initially powered up and a sequence of request vectors $s \in (\{0, 1\}^m)^k$ determine a sequence of annotated decompositions

$$\langle \lambda h. h \parallel \langle (H_2 h_{|h|-1}) s_{|h|-1} \rangle \rangle^{k-1} \langle H_0 t \rangle \in \dot{\mathbb{A}}^k$$

derived from t by Equation 12.28 and Equation 12.40, so that the metric Q parameterized by the decomposition appropriate to each request in the sequence is evaluated accordingly by

$$(Q^* (\lambda h. h \parallel \langle (H_2 h_{|h|-1}) s_{|h|-1} \rangle \rangle^{k-1} \langle H_0 t \rangle) \Delta s \in \mathbb{R}^k.$$

As a property of an arbiter, the result should not really depend on the length k of the sequence used to evaluate the metric, so we might regard

$$\frac{1}{k} \sum (Q^* (\lambda h. h \parallel \langle (H_2 h_{|h|-1}) s_{|h|-1} \rangle \rangle^{k-1} \langle H_0 t \rangle) \Delta s \in \mathbb{R}$$

as the preferable form for purposes of comparison. Then a tuple $v = (r, c, l)$ of request probabilities and locality parameters as proposed in Section 12.4 leads to the concept of an expectation $(\vec{\Theta}_n v) (Q, t) \in \mathbb{R}$ of a metric Q on a decomposition t with respect to an access pattern v in terms of a function

$$\vec{\Theta}_n v : (\dot{\mathbb{A}} \rightarrow (\{0, 1\}^n \rightarrow \mathbb{R})) \times \mathbb{A} \rightarrow \mathbb{R}$$

defined based on Equation 12.42 by

$$\bar{\Theta}_n v = \lambda(Q, t) \cdot \lim_{k \rightarrow \infty} \sum_{s \in (\{0,1\}^n)^k} ((\Theta_n^k v) s) \frac{1}{k} \sum (Q^* (\lambda h \parallel \langle (H_2 h_{|h|-1}) s_{|h|-1} \rangle)^{k-1} \langle H_0 t \rangle) \triangle s. \quad (12.47)$$

12.5.2 Optimization

Whereas Equation 12.47 may be useful for inquiring about a proposed decomposition $t \in \mathbb{A}$, a better question would be that of how to propose one. For a known input arity $n \in \mathbb{N}$, there is no obstacle in principle to obtaining the optimum decomposition with respect to a metric Q and an access pattern v by searching the set of decompositions $\nabla_a n \subset \mathbb{A}$ (Equation 12.24), which is always finite, or sampling it to the extent time permits, provided the limit in Equation 12.47 exists. Formally we can express the optimum decomposition $(\bar{\Theta}_n v) Q \in \mathbb{A}$ as that which minimizes the metric Q given v in terms of a function

$$\hat{\Theta}_n : [0, 1]^n \times ([0, 1]^n)^n \times [0, 1]^n \rightarrow ((\mathbb{A} \rightarrow (\{0, 1\}^* \rightarrow \mathbb{R})) \rightarrow \mathbb{A})$$

defined by

$$\hat{\Theta}_n(v) = \lambda Q. (\lambda(m, t). t) \min \{(m, t) \in \mathbb{R} \times \nabla_a n \mid m = (\bar{\Theta}_n v) (Q, t)\}.$$

Decomposition strategies

A decomposition strategy \mathcal{U}_a fit for Equation 12.26 would generalize this result to multiple arities by way of a family of access patterns $v_n \in [0, 1]^n \times ([0, 1]^n)^n \times [0, 1]^n$ determined empirically or otherwise, and a fixed choice of the metric Q .

$$\mathcal{U}_a = \lambda n. (\hat{\Theta}_n v_n) Q$$

Example of a metric

Because this chapter long enough already, we conclude with only the simplest “hello world” example of an arbiter metric and save the more interesting examples for Appendix E (also because they depend on material from Appendix C).

Everyone knows the meaning of fairness, so what could go wrong with putting a number to it, and what could be fairer than an arbiter that grants all requests with equal probability, favoring none? If not all arbiters can be perfectly fair, then one is fairer than another when its grant probabilities are closer to being equal than those of the other. Hence the metric should quantify in some way the dispersion of grant probabilities about a common value. For a decomposition $t \in \mathbb{A}$ and a request vector $r \in \{0, 1\}^n$, the first grant probabilities are given conveniently by $(\dot{H}_1 t) r \in [0, 1]^n$ in terms of the incremental transfer function, which reduces to

$$h = ((\dot{H}_1 t) r) \uparrow \mathbb{R} - \{0\}$$

if we ignore the necessarily zero-valued grant probabilities due to requests not made. Then a standard statistical measure of dispersion

$$\sum (\lambda u. (u - \frac{1}{|h|} \sum h)^2)^* h$$

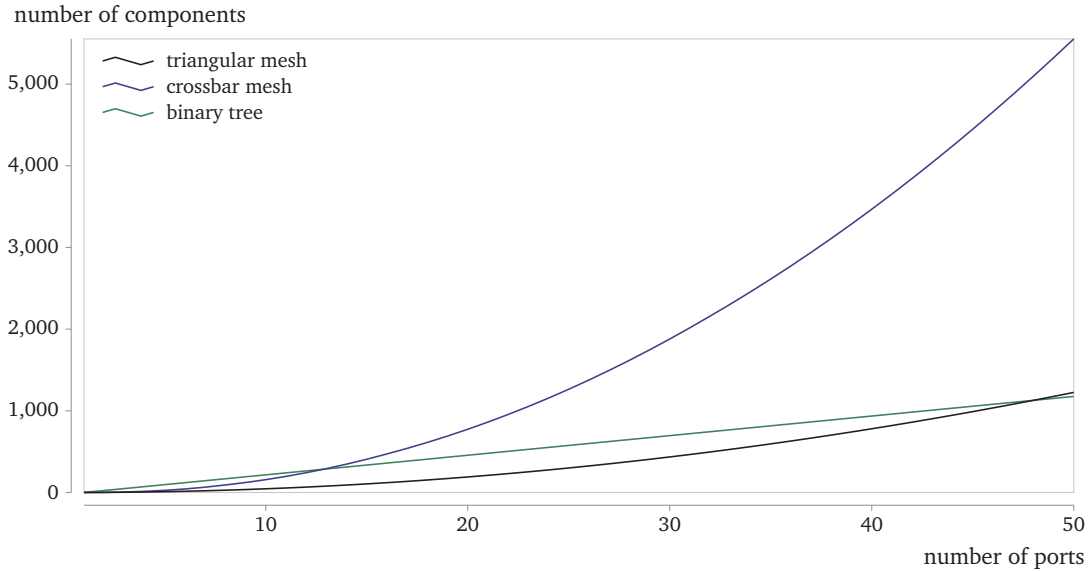


Figure 12.13: Plots of component counts of three families of arbiters illustrate quadratic spatial complexity with the number of ports for meshes and linear for trees.

(i.e., the **variance**) would seem to do the trick, unless perhaps the dispersion should also be normalized with respect to the number of grants being dispersed, as in the metric

$$Q_{hw} = \lambda t. \lambda r. \left(\lambda h. \frac{1}{\delta_0^{|h|} + |h|} \sum (\lambda u. (u - \frac{1}{|h|} \sum h)^2) * h \right) (((\dot{H}_1 t) r) \uparrow \mathbb{R} - \{0\}).$$

On the other hand, the metric Q_{hw} is open to various criticisms as a measure of fairness. For one thing, an arbiter that grants all requests with equal probability but consistently takes longer for some ports than others might not deserve to be called fair, so maybe the critical path metric developed in Section E.2 should factor into it. For another thing, the latency even then is theoretically unbounded due to metastability in the presence of contention [49, 51, 174, 191], which this metric completely ignores. Maybe the contention metric developed in Section E.1 should be incorporated as well, and undoubtedly further embellishments are possible. The takeaway should be that despite the air of mathematical rigor, any choice of a metric is ultimately a judgment call with no guarantee that it captures the quality it purports to measure.

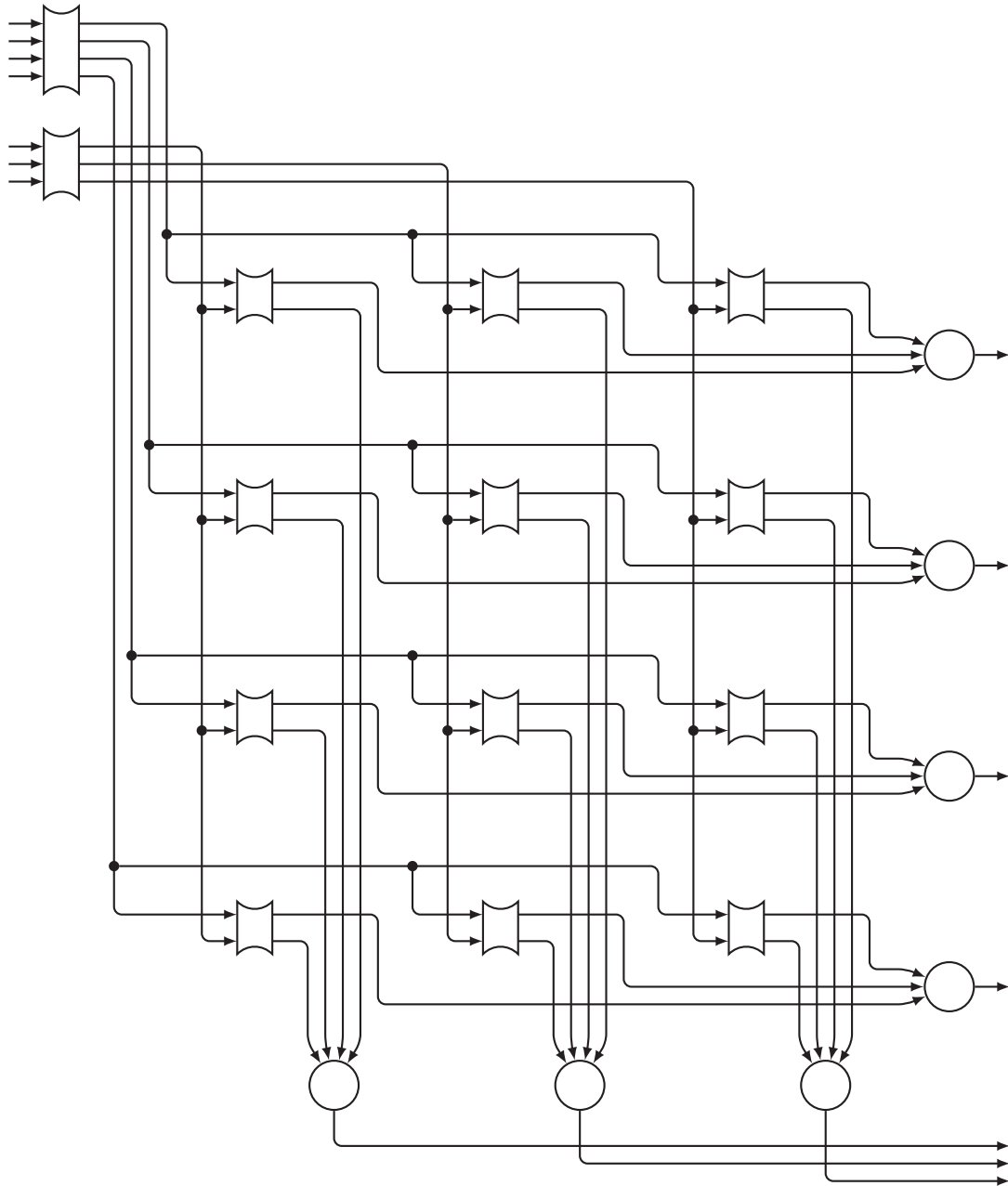


Figure 12.14: A balanced crossbar arbiter is either a primitive or has a front end stage consisting of two smaller balanced crossbar arbiters with equal or nearly equal arities connected to an array of primitive arbiters.

Rear bit

1. Draw circuit diagrams illustrating the broadcast zone examples on page 372. How many components do they need in addition to the primitive arbiters?
2. What process combinator expression $p \in \mathbb{D}$ has the Petri net model in Figure 12.5? What environment $e \in \mathbb{D}$ would make $\mathbf{env}(p, e)$ a better specification for the circuit in Figure 12.4, and what would be better about it?
3. Write a program to implement an arbiter decomposition strategy \mathcal{U}_a that generates a mesh for small arities, a tree for medium sized arities, and a token ring for large ones depending on compile-time constants. What would be a plausible use case?
4. Write out Equation 12.41 the hard way (with no fancy notation) for the two cases $n = 2$ and $n = 3$. How many terms are there on the left sides?
5. (for math geniuses only) Is there a more sophisticated alternative to Equation 12.46 that directly yields an exact analytical solution for all of $\tilde{\Theta}_n(r, c, l)$ consistent with Equation 12.43, Equation 12.44, and Equation 12.45 by construction?
6. With regard to the **crossbar arbiter** in Figure 12.14:
 - a) What specific family of decompositions $t \in \mathbb{A}$ precisely expresses arbiters of this form? (hint: meshes with particular broadcast zones)
 - b) If the two-dimensional array of primitive arbiters visible in the figure were generalized to a k -dimensional array of k -way arbiters, would the crossbar arbiter still be expressible as a mesh? Would it still be an arbiter?
 - c) How many primitive components of any type are there in the average critical path as a function of the number of ports n ? (hint: less than linearly many but more than logarithmic; assume a power of two ports if easier)
7. Write recurrences expressing the total number of primitive components as a function of the number of ports n for triangular mesh arbiters, balanced crossbar arbiters, and binary trees. What are the crossover points in Figure 12.13?
8. An alternative dendriform arbiter proposed in [196] allows each node to serve repeated requests from its descendants without having to issue any release to its parent until it is idle, thereby achieving similar cache-like effects to a token ring.
 - a) What process combinator expression specifies this type of arbiter tree node?
 - b) What delay insensitive circuit design would implement it?
 - c) How could the definitions of Ω_a , \mathbb{A} and H_0 be generalized to accommodate it?
 - d) How should the incremental transfer function \dot{H}_1 , the token distribution function \hat{G} , and the request propagation function \hat{R} be generalized accordingly?

The oldest, shortest words – “yes” and “no” – are those which require the most thought.

Pythagoras

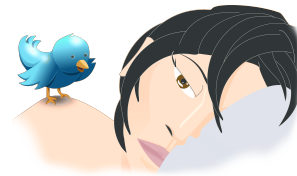
CHAPTER

13

PUTTING THE WORD OUT

At this juncture, much of the preparation for the discussion of general high level circuit synthesis to follow in Part IV is achieved, and readers who have mastered the material in previous chapters can be commended for their persistence. However, one small but important topic could still do with a good mulling over. The problem of conveying information from one place to another in a DI circuit is inherent in any significant project, and is best approached systematically.

The road map for this chapter is as follows. As an aid to understanding the rest, [Section 13.1](#) presents a crash course on some issues pertaining to delay insensitive communication that might not be obvious. Following that, [Section 13.2](#) gives an algorithmic construction of an encoder for any delay insensitive code. The inverse problem of decoding rises and shines in [Section 13.3](#), which generalizes the multidimensional sparse decision wait, and a closely related construction for completion detectors follows in [Section 13.4](#). Finally, [Section 13.5](#) exhibits a class of circuits for transcoding between any two delay insensitive codes.



13.1 Pep talk

The only data communication needed up to this point hardly merits the term: acknowledgments and completion detection signals are limited to communicating that an event has occurred. The situation changes when other types of data are involved. The need to communicate binary, numeric, and more general types of data motivates various solutions discussed in this section.

13.1.1 A two-wire protocol

An immediate difficulty arises even for something as simple as providing a yes or no answer in response to a request, which is that a signal carried by a wire in a DI circuit is unable to distinguish

between the two alternatives. This proposition may be counterintuitive to a reader accustomed to thinking of two-valued logic circuitry, but is inescapable if one envisions a technology of signals mediated by pulses, particles, projectiles, *etc.*, given the technology independence of DI design. A reader unaccustomed to thinking of logic levels has less to unlearn.

If the communication is to be delay insensitive, then at least two wires are needed. The sending and the receiving parts of the circuit would then abide by a protocol identifying a yes answer with a signal on one of the wires and a no with a signal on the other.

13.1.2 1-hot codes

Whereas we may have dodged a bullet above, further difficulties are sure to follow. If the information to be conveyed represents a character of a text message, does delay insensitive communication demand a separate wire for each letter of the alphabet? For very small alphabets, this convention might be feasible, and would be called a **1-hot** code. This terminology should be familiar from the mention of completion detecting buses in [Section 10.3](#). A 1-hot code becomes quite infeasible however for larger alphabets, such as the set of all 128, 172 unicode characters at this writing [63], not to mention for numeric data, because the space needed for that number of wires is prohibitive.

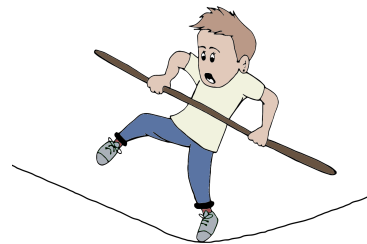
13.1.3 Dual rail codes

Fortunately, the two-wire protocol for binary data described in [Section 13.1.1](#) admits other generalizations than the 1-hot code that cope more effectively with other data communication applications while maintaining delay insensitivity. One generalization, mentioned briefly for motivation in [Section 10.4](#), is the **dual rail** code, and is particularly suitable for numeric data. In a dual rail code, an m bit number is encoded simply by replicating the two-wire solution m times, with one pair of wires for each bit. Arithmetic units operating on numbers in this format are straightforward to design based on conventional algorithms [223]. Dual rail codes are important enough to deserve special consideration in [Chapter 14](#).

The only drawback of dual rail codes is that a cost of two lines per bit is higher than necessary, albeit a considerable improvement on 1-hot coding. How is it possible that two bus lines per bit are excessive while one line per bit is insufficient? Let a **code word** be defined informally for the moment as a unit of information communicable by a given code. An m -bit dual rail code takes a bus with $2m$ wires to transmit one of 2^m possible code words at a time, with each code word representing some number from 0 through $2^m - 1$, but a bus having $2m$ wires would allow more than 2^m ways to encode a word on it if we cared to use them all. Identifying each possibility with a number would allow a wider range of numbers to be communicated via the bus (*i.e.*, more than we could count using numbers with only m bits), so in an information theoretic sense each bit requires something less than two bus lines when the bus is used at full capacity. For the mathematically inclined, that would be

$$2m / \log_2 N$$

bus lines per bit, where the number N , called the **code size** hereafter, is the number of words communicable in the code and $2m$ is the number of lines in the bus.



13.1.4 Constant weight codes

More efficient codes than dual rail treat the bus not as m separate pairs of lines with one pair for each bit, but as a homogenous set of $n = 2m$ lines. With n lines on the bus, there are 2^n subsets of bus lines and therefore 2^n different ways to send a set of signals concurrently. If each subset were identified with a unique code word, we could have a code size of $N = 2^n$, but contrary to expectations based on Section 13.1.1 this code would imply a need for no more than one bus line per bit. Although we can do better than dual rail codes, we can not do that much better.

To see why the ideal of one bus line per bit is unattainable, consider the dubious case of a code with eight words and a bus with three lines numbered from 0 through 2. Each code word is communicated by concurrent signals on a particular subset of the lines. Let each subset be identified by its line numbers. Then there are eight sets

$$\mathcal{P}(\{0, 1, 2\}) = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$$

with one for each code word.

Suppose we try to use this code in practice. When the receiver receives no signals, is it because the sender has communicated the code word associated with \emptyset , or because the bus is idle? There is no resolution compatible with delay insensitivity other than to exclude \emptyset from the set of valid encodings, which already raises the cost per bit above one bus line. However, disallowing empty encodings does not solve the problem completely. If the receiver receives a signal on line number 1, should it recognize it as the encoding of the word associated with the subset $\{1\}$, or should it allow for the possibility that the rest of either $\{0, 1\}$, $\{1, 2\}$ or $\{0, 1, 2\}$ is still in transit? The only resolution is to restrict the set of valid encodings further. Exclusion of empty encodings is a consequence of the more general condition that in a delay insensitive code, *no encoding may be a proper subset of any other*.

A fancy designation for a set of sets meeting this condition is as an **antichain** [130]. For example, if the only valid encodings were $\{0, 1\}$, $\{0, 2\}$, and $\{1, 2\}$, they would form an antichain and the uncertainty noted above would be eliminated. Having received a signal on line number 1, the receiver would be obliged to wait for exactly one more signal, which would determine whether the transmitted set was meant to be $\{0, 1\}$ or $\{1, 2\}$.



As this example suggests, a sufficient way to construct an antichain is to restrict all sets to a constant cardinality $k > 0$. The case of $k = 1$ reduces to a 1-hot code. The current example would be called a 2-of-3 code, because there are two lines used in any encoding on a three line bus. In general, a code fitting this pattern is called a k -of- n code, or alternatively a **constant weight** code.

Being the number of subsets of cardinality k of a set of cardinality n , the code size for a k -of- n code is always the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

shown diagrammatically in Figure 13.1. This number is maximized for a given n when k is close to $n/2$, and maximizing the code size for a given bus width maximizes the rate of communication, or what is conventionally called the **channel capacity** (a term to be used only informally for our purposes). Codes satisfying $k = n/2$ when n is even or $k \in \{\lfloor n/2 \rfloor, \lceil n/2 \rceil\}$ when n is odd are often called **balanced codes** [145, 277], but also come under the headings of **DC-free** and **RDS-minimizing** codes as special cases thereof [109, 122].

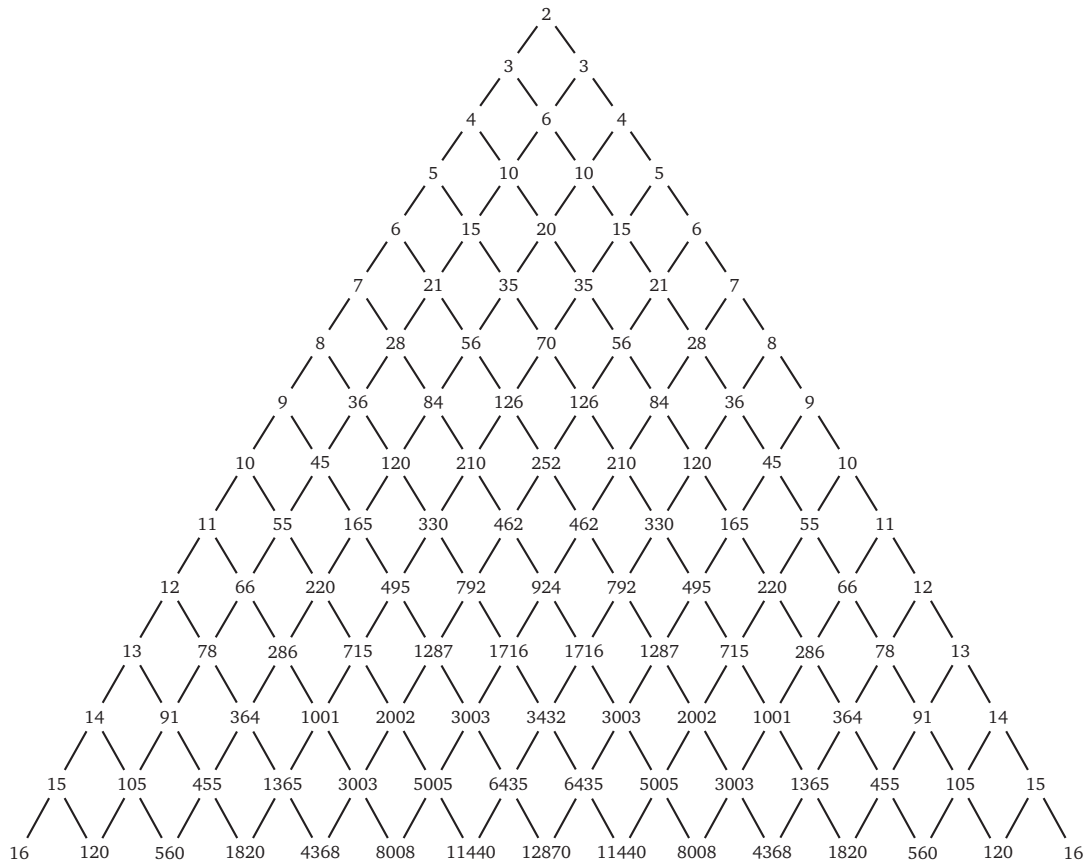


Figure 13.1: k -of- n code sizes for n ranging from 2 to 16 and k ranging from 1 to $n - 1$ are binomial coefficients, shown here in a segment of Pascal's triangle.

13.1.5 General delay insensitive codes

A constant cardinality k is a sufficient but not a necessary condition for an antichain, whereas any antichain whatsoever over a finite set determines a delay insensitive code. It is therefore appropriate to ask whether better ones than k -of- n codes are possible. That is, can the channel capacity be optimized by tweaking the code to have fewer than k signals in some of the encodings and more than k in others provided that the encodings still form an antichain? Other things being equal, this question is settled by Sperner's theorem, a standard result in combinatorics, which implies that no code size greater than that of a balanced k -of- n code is possible for a bus with n lines [130].

Nevertheless, balanced codes might not be the only kind of codes ever worth using. Trading channel capacity for reduced circuit complexity can be a perfectly valid engineering option in some circumstances. For example, there are no known algorithms other than look-up tables for doing arithmetic directly on numbers in optimal k -of- n format, hence the justification for dual rail codes. Furthermore, "other things" are not equal in the presence of noise on the channel insofar as error correction may incur cost or performance penalties that affect the right choice of a code. Any error

correcting code works ultimately by prohibiting the use of certain patterns as valid encodings, so that an error is identified whenever one of them occurs. This consideration could motivate the use of a subset of the full balanced code, or even a non-constant weight code depending on the peculiarities of the error correction mechanism. Finally, a code may have fewer words than possible for its bus width simply because the number of semantically useful messages an application needs to communicate turns out to be some number other than a binomial coefficient.

It should be mentioned in passing that balanced error correcting codes pose unique challenges, and the situation becomes more problematic when delay insensitivity is involved [155, 188, 256, 299]. Spurious events generating more than k signals in a k -of- n code are likely to induce divergence in the receiver, and events that suppress any signals necessarily cause deadlock. Theoretically, errors that maintain invariance of the code weight would be amenable to detection and correction at the level of a DI circuit, but current technologies do not oblige signals to observe any such conservation principle.

To summarize, the most general class of delay insensitive codes includes codes that are balanced or unbalanced, with constant weight or not, as well as any arbitrary subset of those, provided their respective sets of encodings form an antichain. All of them may well have their uses even if some of them are less efficient. We therefore target this class in the remainder of this chapter for our decoders, completion detectors, encoders, and transcoders.

13.1.6 Terminology

To discuss them more succinctly henceforth, we model delay insensitive codes as finite non-empty antichains

$$c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$$

which is to say that they satisfy

$$\forall v, w \in c. v \not\subseteq w$$

as noted in Section 13.1.4, and we refer to the model c itself as a code. We also refer to the members w of a code c simply as its code words rather than as the encodings thereof, dropping the distinction maintained above. In addition, a member i of a code word w is called a **symbol** when there is a need to refer to it specifically. The set of symbols $\bigcup c$ associated with a code c is called its **alphabet**, and as a matter of convenience we insist on the alphabet of any code c forming a consecutive set containing zero.

$$|\bigcup c| = 1 + \max \bigcup c \tag{13.1}$$

13.2 Encoders

This section describes encoders because they are the simplest family of circuits in this chapter and a prerequisite to those that follow. An encoder transforms a stream of 1-hot coded input words into output words in some arbitrary code, usually with fewer symbols on the output side and therefore a narrower bus. The design in Figure 13.2, which generalizes to any encoder, never needs more than a FORK network, a permutation network, and a MERGE network. An enumeration of the code words, such as

$$\begin{aligned} & \{\{0, 1, 2\}, \{0, 1, 3\}, \{0, 1, 4\}, \{0, 1, 5\}, \{0, 2, 3\}, \\ & \{0, 2, 4\}, \{0, 2, 5\}, \{0, 3, 4\}, \{0, 3, 5\}, \{0, 4, 5\}, \\ & \{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \\ & \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\} \end{aligned}$$

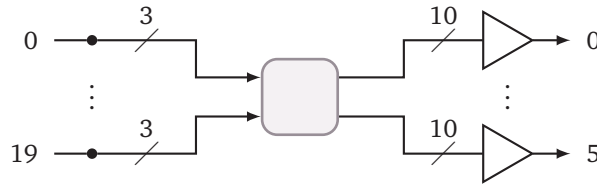


Figure 13.2: An example of an encoder interfacing a 20-line 1-hot channel with a 3-of-6 coded channel has twenty 3-way FORK networks, a permutation network, and six 10-way MERGE networks.

for the example of a 3-of-6 code, should tell us everything necessary to build the encoder, and it does. If the MERGE networks are numbered from 0 to 5, then each code word w indicates the need for a FORK with $|w| = 3$ outputs, and the members of w identify the destination MERGE numbers of the FORK outputs via the permutation network. There is probably even a formula for the set

$$\begin{aligned} & \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, & & \{0, 1, 2, 3, 10, 11, 12, 13, 14, 15\}, \\ & \{0, 4, 5, 6, 10, 11, 12, 16, 17, 18\}, & & \{1, 4, 7, 8, 10, 13, 14, 16, 17, 19\}, \\ & \{2, 5, 7, 9, 11, 13, 15, 16, 18, 19\}, & & \{3, 6, 8, 9, 12, 14, 15, 17, 18, 19\} \end{aligned}$$

enumerating the source FORK numbers ranging from 0 to 19 for each 10-way MERGE if such a thing is of any use.

Another way of thinking about an encoder, possibly more intuitive from a computer architect's point of view, is as a read-only memory with a 1-hot address bus in and a data bus out. The number of memory locations is equal to the number of address lines, and each memory location stores a code word. The alphabet of the stored code words determines the data bus width, which has a value of 6 in this example.

This way of thinking about encoders suggests a minor enhancement to make them slightly more interesting. Just as there is no need for the words in a memory to be mutually distinct, there is no need to limit the number of memory locations to the number of possible code words. It might be useful for application specific reasons for an encoder to implement a look-up table containing duplicate entries. To specify an encoder having this capability, it is best to parameterize a function $\text{EC} : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ by a list

$$c \in \mathcal{P}(\mathbb{N})^*$$

of the code words $w \in \mathcal{R}(c)$ in order of their addresses in the encoder (or memory) $\text{EC}(c) \in \mathbb{H}$. The code carried by the output data bus is technically $\mathcal{R}(c) \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ according to the terminology proposed in Section 13.1.6.

Although a list of code words is more appropriate for an encoder specification than a set because it makes any desired ordering expressible and enables duplicate entries, the analogous condition to Equation 13.1 is worth requiring of it nevertheless to simplify the derivations to follow, and doing so incurs no loss of generality.

$$|\bigcup \mathcal{R}(c)| = 1 + \max \bigcup \mathcal{R}(c) \quad (13.2)$$

That is, we can always assume that the set of data line numbers forms a consecutive sequence starting with zero, or equivalently that every symbol appears in at least one memory location or table entry. The alternative would imply a data line that is never used and therefore would be

better to eliminate from the specification. In practice, the code words $\mathcal{R}(c)$ should normally form an antichain as well if the encoder is to be useful for anything, but technically only their non-emptiness is required for the forthcoming constructions to yield valid results.

$$\emptyset \notin \mathcal{R}(c)$$

The rest of this section focuses constructing the encoder generating function EC envisioned above, first limited to the basic encoder depicted in [Figure 13.2](#), and then generalized by two possible optimizations.

13.2.1 Basic

For the basic encoder, there are $|c|$ multi-way FORK networks connected to $|\bigcup \mathcal{R}(c)|$ multi-way MERGE networks as shown in [Figure 13.2](#), where $c \in \mathcal{P}(\mathbb{N})^*$ is the list of code words. The output arity of the j -th FORK is simply $|c_j|$, the cardinality of the j -th code word in the list, but the input arity of the i -th MERGE network must match the number of code words with an i in them. It is therefore evident from the total number of i 's appearing in all terms throughout c ,

$$|(\overset{\circ}{b} c) \uparrow \{i\}|$$

also expressible as $|a \uparrow \{i\}|$ with $a = \overset{\circ}{b} c$, for all symbols i in the alphabet $\mathcal{R}(a)$. A list of the MERGE input arities in order of their associated output alphabet symbols is expressible as $A_0 c$ with the function $A_0 : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{N}^*$ defined by

$$A_0 = (\lambda a. (\lambda i. |a \uparrow \{i\}|)^* \mathcal{R}(a)^{\circ-1}) \circ \overset{\circ}{b}. \quad (13.3)$$

The permutation network between the FORK and the MERGE networks in [Figure 13.2](#) should connect f -th FORK to the m -th MERGE for each $m \in c_f$. The position of any terminal on this MERGE relative to all other MERGE input terminals must be at least

$$|(\overset{\circ}{b} c) \uparrow \mathcal{R}(t_m)|$$

the total number of input terminals due to any MERGE numbered less than m , in addition to an offset

$$|(\overset{\circ}{b}(c \uparrow f)) \uparrow \{m\}|$$

due to all inputs on the same MERGE connected to any FORK numbered less than f . These conditions determine a permutation $A_1 c \in \mathbb{N}^*$ with $A_1 : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{N}^*$ given by

$$A_1 = \lambda c. \overset{\circ}{b} (\lambda f. (\mu \lambda m. |(\overset{\circ}{b} c) \uparrow \mathcal{R}(t_m)| + |(\overset{\circ}{b}(c \uparrow f)) \uparrow \{m\}|) c_f)^* \iota_{|c|} \quad (13.4)$$

and therefore a general specification of the encoder $\text{EC}_0 c \in \mathbb{H}$ in [Figure 13.2](#) for a function $\text{EC}_0 : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ given by

$$\text{EC}_0(c) = (\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK} |f|)^* c \xrightarrow{A_1 c} (\mathcal{F} \mathbf{R}) \text{MERGE}^* A_0 c \quad (13.5)$$

denoted as such in light of a couple of optimizations to follow.

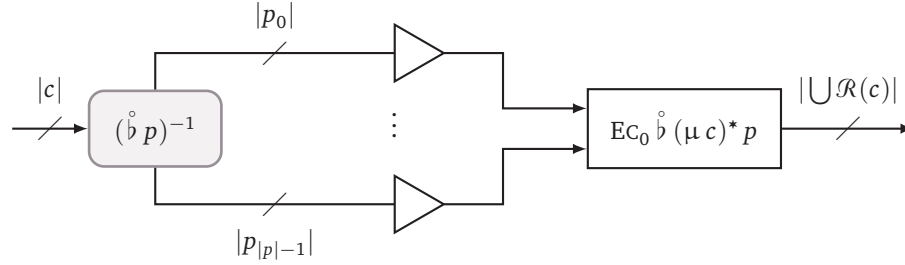


Figure 13.3: A front optimized encoder $EC_1(c)$ uses a list $p = ((\pi c) \mathcal{D}(c))^{\circ-1} \in \mathcal{P}(\mathbb{N})^*$ of equivalence classes of input code word indices inferred from the list $c \in \mathcal{P}(\mathbb{N})^*$ of code words, with one basic encoder input for each class.

13.2.2 Front optimized

One way of improving on the basic encoder generating function EC_0 in Equation 13.5 is by taking advantage of duplicates in the list c of code words. If two words $c_a = c_b$ are identical for distinct indices $a \neq b$, then a signal to the a -th input line causes same set of output lines numbered $m \in c_a$ to transmit signals as an input signal to the b -th line does. It would make no observable difference to put the a -th and b -th input lines through a MERGE and then to treat the MERGE output as a single input to a smaller encoder. Compared to implementing the same specification as a basic encoder, this optimization would save the cost of at least one front end FORK network, and also would reduce the input arity by at least one on $|c_a|$ of the back end MERGE networks (hence eliminating a MERGE primitive from each of them).

To gain this advantage, we first construct the lexicographically ordered list of equivalence classes

$$p = ((\pi c) \mathcal{D}(c))^{\circ-1} \in \mathcal{P}(\mathbb{N})^*$$

of input word indices such that any two input word indices $a, b \in \mathcal{D}(c)$ satisfying $c_a = c_b$ are members of the same class p_i for some $0 \leq i < |p| \leq |c|$ by Equation 6.6 (using the list c implicitly as a function here, with πc short for $\pi \lambda a. c_a$). In the improved encoder, the i -th of $|p|$ MERGE networks in parallel has an input arity of $|p_i|$ and connects to all lines numbered $a \in p_i$ from the external input bus, or reduces to a single wire by Equation 9.19 if there is only one such $a \in p_i$. With p_i identifying the set of source input indices to the i -th MERGE for each i , these connections require an input permutation network described by the permutation

$$(\overset{\circ}{b} p)^{-1}$$

between the input bus and the MERGE network to maintain the original order of inputs visible externally.

The outputs from the MERGE network connect to an internal basic encoder having $|p|$ inputs as shown in Figure 13.3, but because $|p|$ is less than $|c|$ if there are any duplicates in c , the internal encoder is specified by an abbreviated list of code words. For consistency with the MERGE network, the i -th term in the abbreviated list must be a code word c_a for some member $a \in p_i$, which could be any member of p_i because c is the same for all of them by construction. Evaluating $(\mu c)^* p$ therefore yields a list of $|p|$ singleton sets of code words, which can be flattened into a list

$$\overset{\circ}{b} (\mu c)^* p$$

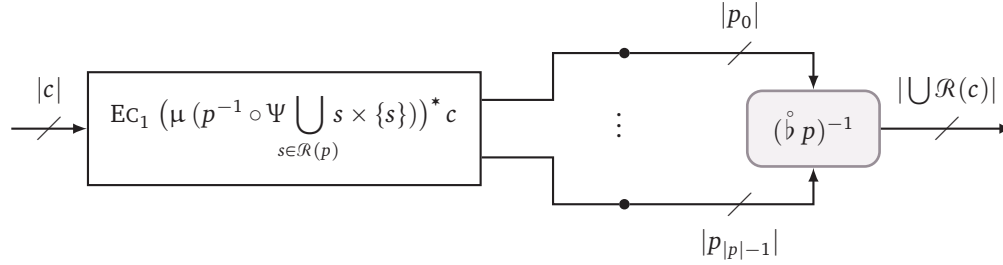


Figure 13.4: A back optimized encoder $EC(c)$ partitions the outputs into equivalence classes $p = A_2 c$ by Equation 13.6 with a FORK network and only one internal encoder output for each class.

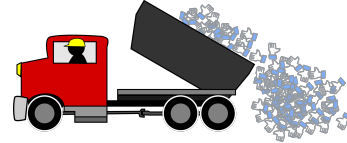
of $|p|$ code words of the required length and order.

These observations lead to the following definition for a function $EC_1 : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ taking a list of code words $c \in \mathcal{P}(\mathbb{N})$ to a front optimized encoder $EC(c) \in \mathbb{H}$.

$$EC_1(c) = (\lambda p. (\overset{\circ}{b} p)^{-1} \times C_{|p|} \langle (\mathcal{F} R) (\lambda m. \text{MERGE } |m|)^* p, EC_0 \overset{\circ}{b} (\mu c)^* p \rangle) ((\pi c) \mathcal{D}(c))^{-1}$$

13.2.3 Back optimized

Another improvement to the basic encoder is possible by taking advantage of specifications wherein two or more symbols always appear together whenever any of them is present in a code word. This condition implies that an output data line numbered m is equivalent to some other line n in that a signal transmitted on either of them indicates that one must also be transmitted on the other. In this case, we achieve the same effect by eliminating the back end MERGE network associated with one of them (e.g., m) and connecting a FORK to the other one (e.g., n) with an output permutation network to route the outputs from the FORK to positions n and m on the data bus. We can also use this technique on a front optimized encoder instead of a basic encoder to obtain one that is optimized at both ends.



To describe the FORK network more precisely, let $a = \bigcup \mathcal{R}(c)$ denote the output alphabet determined by the list of code words $c \in \mathcal{P}(a)^*$ and let $o \in a$ denote an output alphabet symbol, so that the set $a - \{o\}$ of alphabet symbols other than o and the list

$$c \uparrow \mathcal{P}(a - \{o\}) \in \mathcal{P}(a)^*$$

of code words not containing o induces a partition

$$(\pi \lambda o. c \uparrow \mathcal{P}(a - \{o\})) a \in \mathcal{P}(\mathcal{P}(a))$$

whereby any two symbols $n, m \in a$ are in the same class only if both are members of exactly the same words. A lexicographically ordered list $p = A_2 c \in \mathcal{P}(a)^*$ of these equivalence classes is obtained by a function $A_2 : \mathcal{P}(\mathbb{N})^* \rightarrow \mathcal{P}(\mathbb{N})^*$ defined as

$$A_2 = \lambda c. (\lambda a. ((\pi \lambda o. c \uparrow \mathcal{P}(a - \{o\})) a)^{\circ^{-1}}) \bigcup \mathcal{R}(c). \tag{13.6}$$

Then for each $0 \leq i < |p|$, there is a FORK with output arity $|p_i|$ in parallel with the others as shown in Figure 13.4, whose output terminals transmit the signals associated with members of p_i . Ordering the output terminals consecutively with respect to the alphabet requires an output permutation network from the FORK network described by the permutation

$$(\overset{\circ}{b} p)^{-1} \in \mathbf{a}^{|a|}.$$

The internal decoder needed by the back optimized form has the same number $|c|$ of inputs as the original specification, but only $|p| \leq |c|$ outputs because of the combined output symbols, so it is described by a list of $|c|$ words v each derived from the corresponding word $w \in \mathcal{R}(c)$. Each symbol $o \in w$ is a member of a class $p_i \in \mathcal{R}(p)$, and the corresponding word v in the list parameterizing the internal encoder is the set of all class indices $i \in \mathcal{D}(p)$ of any classes containing symbols $o \in w$.

A formal expression of this idea starts with a class $s \in \mathcal{R}(p)$ determining a set of pairs $s \times \{s\}$ whose left is a member of the class (that is, a symbol from the alphabet) and whose right is the whole class. The union of all such sets determines a function

$$\Psi \bigcup_{s \in \mathcal{R}(p)} s \times \{s\}$$

by Equation 6.1 mapping any symbol to the class containing it, which determines a function

$$p^{-1} \circ \Psi \bigcup_{s \in \mathcal{R}(p)} s \times \{s\}$$

taking any symbol to the ordinal of its class relative to p . The word v described above comes from mapping this function over a word $w \in \mathcal{R}(c)$

$$v = (\mu (p^{-1} \circ \Psi \bigcup_{s \in \mathcal{R}(p)} s \times \{s\})) w$$

which implies $|v| < |w|$ whenever any two or more symbols in w belong to the same class $s \in \mathcal{R}(p)$. The list of words parameterizing the internal encoder follows from mapping this function over all words in the original specification c .

$$(\mu (p^{-1} \circ \Psi \bigcup_{s \in \mathcal{R}(p)} s \times \{s\}))^* c$$

Putting the internal encoder, the FORK network, and the permutation network together based on the foregoing leads to back optimized encoder $\text{EC}(c) \in \mathbb{H}$ in terms of a function $\text{EC} : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ defined as follows

$$\text{EC}(c) = (\lambda p. \mathbf{C}_{|p|} \langle \text{EC}_1 (\mu (p^{-1} \circ \Psi \bigcup_{s \in \mathcal{R}(p)} s \times \{s\}))^* c, (\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK } |f|)^* p \rangle \times (\overset{\circ}{b} p)^{-1}) A_2 c \quad (13.7)$$

which is also front optimized by the function EC_1 defined previously.

13.3 Decoders

This section is concerned with constructing delay insensitive decoders algorithmically based on a specification $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ for a code as proposed in Section 13.1.6 (N.B., not a list of code words as

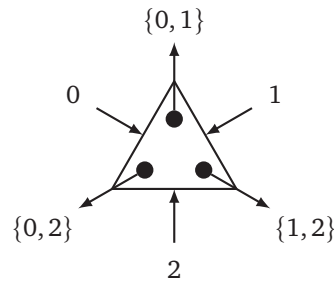


Figure 13.5: A TRIA has a schematic symbol resembling a decision wait. Concurrent inputs to any two sides cause an output from their common corner.

in the case of an encoder). A decoder is defined for our purposes as a circuit that interfaces an input bus carrying an arbitrary delay insensitive code c with an output bus carrying a 1-hot code such that each code word received via the input bus results in a distinct code word transmitted via the output bus (that is, one individual output signal). A decoder for a code c therefore has $|\bigcup c|$ inputs and $|c|$ outputs. The 1-hot output bus is intuitively appropriate for a decoder because it unpacks the code to the point of uniquely identifying each word, hence leaving nothing further about it to be decoded. Decoders are conceptually important because they provide an assurance that the encoded information is completely recoverable.

An expression $Dc(c) \in \mathbb{H}$ denoting a decoder in terms of a generating function $Dc : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$ and a code specification c is the main goal for this section. Other relevant conventions are to associate the i -th input symbol $i \in \bigcup c$ with the i -th input terminal on $Dc(c)$, and the j -th output terminal on $Dc(c)$ with $c^{\circ-1} j$, the j -th code word in lexicographic order. Any other arbitrary terminal ordering is always achievable in practice through additional permutation networks. We begin with a solution in Section 13.3.1 that always works when all else fails, and then consider possible shortcuts applicable to codes of particular forms thereafter (*e.g.*, factorable and partitionable codes). A general recurrence incorporating any of these methods to construct a decoder for a given code is proposed in Section 13.3.5.

13.3.1 Basic

To revisit the example of the 2-of-3 decoder in a more concrete form, we would have a code consisting of three words on a three-symbol alphabet

$$c = \{\{0, 1\}, \{0, 2\}, \{1, 2\}\}$$

and therefore a decoder with three inputs and three outputs. The first input would be for the symbol 0, the next for the symbol 1, and the last for the symbol 2. The first output would correspond to the code word $\{0, 1\}$, this being the first code word in lexicographic order. This output would be signaled by the decoder whenever it receives the inputs on line 0 and line 1. The next output would be for the code word $\{0, 2\}$, and the last for $\{1, 2\}$. A 2-of-3 decoder is called a TRIA in [222], one of the few decoders anyone has considered it worthwhile to name. The schematic symbol for a TRIA with terminals labeled according to the current convention is shown in Figure 13.5.

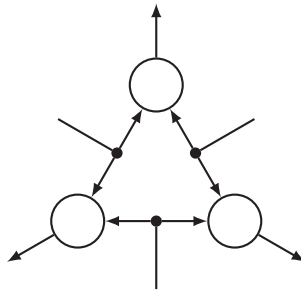


Figure 13.6: an incorrect implementation of a TRIA

The TRIA suggests an implementation along the lines of [Figure 13.6](#). This implementation is deliberately incorrect but is worth contemplating momentarily because it motivates a correct one to follow. Although an output signal emerges from the expected JOIN the first time any two inputs are applied, a couple of stale input signals forked to the other two are waiting to cause trouble the next time around (cf. [Figure 2.5](#)). If the previously unused input is applied next, then the other two outputs emerge concurrently, which is bad enough, but if either of the previously used inputs is applied again, nothing more can be predicted because the whole system diverges.

If only the signals sent to the other two JOIN primitives during the first use could be unsent to them somehow after they have been detected by the one that needs them, then everything would be fine. For example, if signals on lines 0 and 1 are received, they are useful to the JOIN associated with the output $\{0, 1\}$ and not needed by the other two. Unsending a signal is problematic to say the least, but the next best thing might be to send yet another signal that effectively resets any device that should not have received it. When signals on lines 0 and 1 arrive, the other two JOIN primitives are both stuck waiting for one on line 2. If the JOIN associated with $\{0, 1\}$ were to send both of them a simulated input signal on line 2, then wait for them to get back to normal, and only then transmit an external acknowledgment, correct decoding would be accomplished.

As far fetched as it may seem, this method of operation is precisely the intuition underlying the design in [Figure 13.7](#), but filling in the details leads to a few complications. Because every input to a front end JOIN could be either real or simulated, it feeds through a MERGE driven by signals originating either locally or externally. The output from each front end JOIN has to feed through a cascade of two SHUNT and TOGGLE combinations, because it may need to be diverted in either of two ways. The first signal to propagate through both stages of its respective cascade forks to the control inputs on two of the others, resetting the corresponding JOIN only after arranging for its output signal to be diverted via the SHUNT. The diverted signals take the alternate route out of the SHUNT and TOGGLE combination and synchronize on the appropriate output stage JOIN as required.

Slicing and dicing

The best thing about this TRIA implementation is that it generalizes to a decoder for any delay insensitive code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$. The pattern of three horizontal slices evident from [Figure 13.7](#) becomes one slice for each of the $|c|$ code words, with each slice containing a MERGE network, a multi-way JOIN in front feeding into a SHUNT cascade ending with a FORK, a TOGGLE array, and another multiway JOIN in the back.

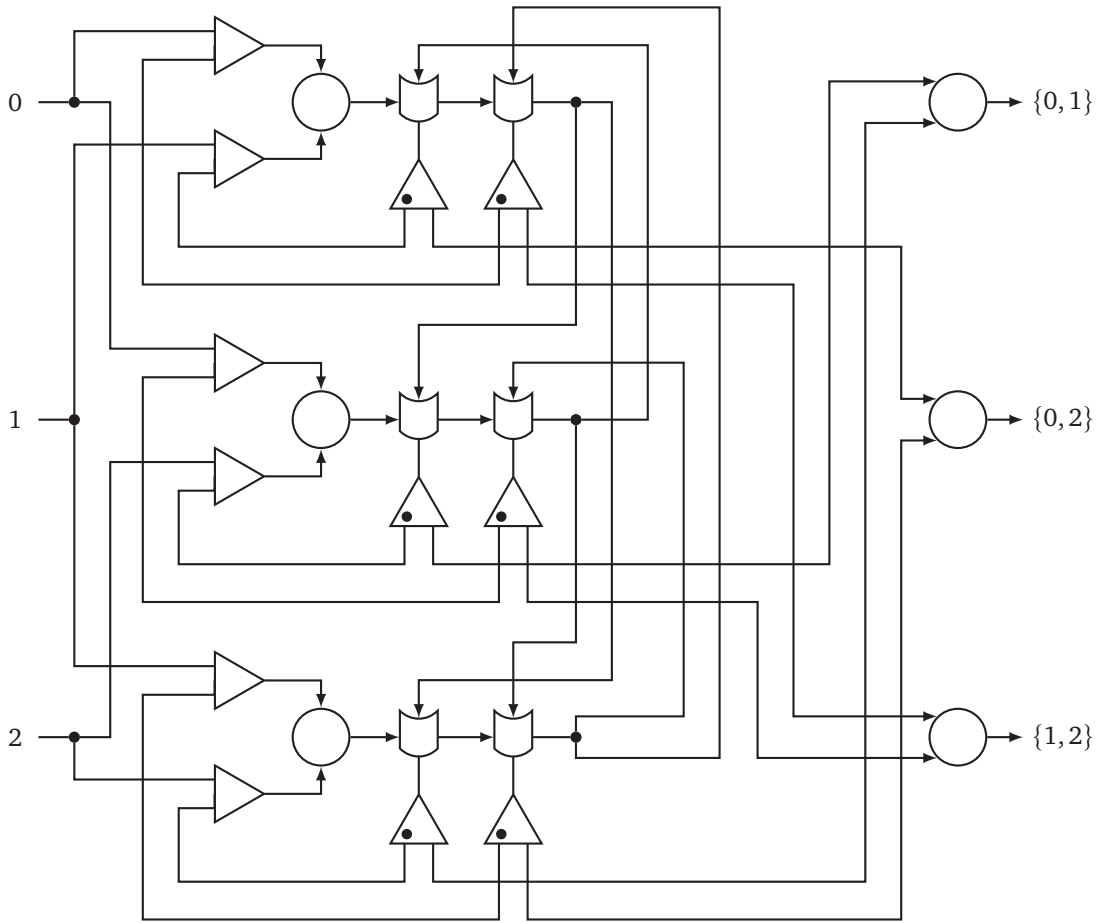
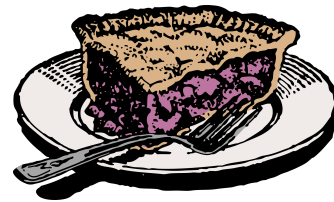


Figure 13.7: A correct TRIA implementation in three slices suggests a general form for any decoder.

The FORK network shown at the left of Figure 13.7 has one input for each of the $|\cup c|$ alphabet symbols regardless of the number of slices, and corresponds roughly to the FORK primitives appearing in Figure 13.6, but the internal FORK primitives in Figure 13.7 have no counterpart in Figure 13.6. The latter are used to broadcast the so called reset signals to the other slices. Their use is based on a design decision favoring performance over cost analogous to that of Figure 10.19. An alternative featuring a sequential communication from one slice to another as in Figure 10.6 is also possible and would eliminate the need for the back end JOIN array. The difference in cost is loosely bound by a constant factor of the cost of the whole decoder, whereas a difference between linear and logarithmic latency in the alphabet cardinality is at stake. Although it is not necessarily always the inferior choice, the alternative is not developed further here.



SHUNT cascade

The rest of this section is about putting a generalization of Figure 13.7 with respect to an arbitrary code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ into writing, starting with the SHUNT cascade that appears in the middle of a typical slice pertaining to a word $w \in c$. Before getting even that far, we have to ascertain the length of the cascade. If the decoder receives another word v containing some of the same symbols as w , then the front JOIN associated with the slice for w receives their common symbols as well and needs resetting. Each reset signal from another slice connects to the control input on one SHUNT in the cascade, so there must be one for each word $v \in c$ that intersects w . Letting $t = (I_0 c) w$ denote the lexicographically ordered list of all other words in c intersecting w according to a function

$$I_0 : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow (\mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})^*)$$

defined in the obvious way by

$$I_0 = \lambda c. \lambda w. (c - \mathcal{P}(\mathbb{N} - w) - \{w\})^{o-1}$$

we have a cascade in $s = |t|$ stages given already by the expression

$$\mathbf{U}((\mathbf{L}\langle \text{SHUNT}, \text{TOGGLE} \rangle \uparrow 1)^s) \downarrow 1$$

where the first output from each SHUNT connects to the first input of the next. The first input to the cascade is the first input to the first SHUNT, and the next s are the reset inputs to the slice. The output from the last SHUNT is rolled from the last position to the top so that the TOGGLE outputs can be separated into two buses of s lines each in $I_1 s$ with $I_1 : \mathbb{N} \rightarrow \mathbb{H}$ defined by

$$I_1 = \lambda s. \mathbf{U}((\mathbf{L}\langle \text{SHUNT}, \text{TOGGLE} \rangle \uparrow 1)^s) \downarrow 1 \times \iota_{2s} \times s \uparrow 1$$

before the SHUNT output rolls back to the bottom.

MERGE network

The next thing to organize is the MERGE network shown at the left of Figure 13.7 interfacing the TOGGLE outputs with the front JOIN in each slice. The first s outputs from the cascade $I_1 s$ come from the first TOGGLE output on each stage and need to be connected to the MERGE inputs. There is in general one multi-way MERGE in the slice for each alphabet symbol in w , which puts them not necessarily into one to one correspondence with the stages of the cascade despite being that way in Figure 13.7. More precisely, the output from the j -th TOGGLE in the cascade should be connected to the k -th MERGE for every symbol $k \in w - t_j$ (via a multi-way FORK of the appropriate output arity), where t_j following the discussion above is the lexicographically j -th word in c that intersects w . In this way, only the additional inputs to the slice that have not already been received as an unwanted side effect of receiving the word $v = t_j$ are fed to the JOIN to reset it, as required.

The easy way to implement most of the interface between the TOGGLE and MERGE networks is to recognize that a block consisting of a FORK network connecting a 1-hot channel to a MERGE network fits the pattern of an encoder according to Section 13.2. The list of code words specifying the encoder is something like

$$(\lambda v. w - v)^* t$$

because each word $w - v$ enumerates the set of destinations in the MERGE network for signals to be sent when the receipt of a word v mandates a reset of the slice for w . The list of words is not exactly

as above because its range might not satisfy Equation 13.1, a condition assumed in the derivation of Equation 13.7, but this issue is resolved by an encoder $\text{EC } e \in \mathbb{H}$ for a list of words

$$e = (\lambda v. (\mu (w - \bigcup \mathcal{R}(t))^\circ) (w - v))^* t \in \mathcal{P}(\mathbb{N})^* \quad (13.8)$$

whose alphabet $a = \bigcup \mathcal{R}(e)$ renumbers the symbols in w that are not members of any $v \in \mathcal{R}(t)$ while preserving their order. If this specification enables any front or back optimizations by Equation 13.7, so much the better.

In addition to the MERGE network built into the encoder, another layer of MERGE inputs needs to be exposed to the FORK network carrying the external inputs shown at the left of Figure 13.7. If the encoder alphabet a has the same cardinality as the word w , then an array of $|a| = |w|$ additional MERGE primitives each having one input connected to an encoder output and the other left exposed is sufficient. However, the word w could also contain symbols not common to any other words v , which would mean $|w|$ exceeds $|a|$ and some of the inputs to the JOIN come directly from the front end FORK network without going through a MERGE. To cover both cases, a block

$$\mathbf{F}_{|a|} \langle \text{EC } e, \mathbf{R}(\text{MERGE}^{|a|} \leftarrow_{\gamma_2^1} \mathbf{I}^{|w-a|}) \rangle$$

has s inputs first to interface with the TOGGLE array, followed by $|w|$ inputs to a combination of $|a|$ half-connected MERGE primitives and $|w - a|$ extra wires to make up the total. It would be helpful for these last $|w|$ inputs to be ordered consistently with the symbols in w corresponding to them whether or not a given symbol appears in any other words v , which is possible by an input permutation network connected to the bottom $|w|$ inputs, as in

$$\mathbf{I}^{|w|} \xrightarrow{p} \mathbf{F}_{|a|} \langle \text{EC } e, \mathbf{R}(\text{MERGE}^{|a|} \leftarrow_{\gamma_2^1} \mathbf{I}^{|w-a|}) \rangle$$

for a permutation $p = (\overset{\circ}{b} \langle a, w - a \rangle)^{-1}$, which we might summarize as $(I_2 w) e \in \mathbb{H}$ for

$$I_2 : \mathcal{P}(\mathbb{N}) \rightarrow (\mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H})$$

defined by

$$I_2 = \lambda w. \lambda e. (\lambda a. (\lambda p. \mathbf{I}^{|w|} \xrightarrow{p} \mathbf{F}_{|a|} \langle \text{EC } e, \mathbf{R}(\text{MERGE}^{|a|} \leftarrow_{\gamma_2^1} \mathbf{I}^{|w-a|}) \rangle) (\overset{\circ}{b} \langle a, w - a \rangle)^{-1}) \bigcup \mathcal{R}(e)$$

or more conveniently as $I_3(w, t) \in \mathbb{H}$ based on Equation 13.8 and a function

$$I_3 : (\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})^*) \rightarrow \mathbb{H}$$

defined by

$$I_3 = \lambda(w, t). (I_2 w) (\lambda v. (\mu (w - \bigcup \mathcal{R}(t))^\circ) (w - v))^* t$$

where it should be noted that its first $|w|$ inputs on the block $I_3(w, t)$ are to the exposed MERGE network or extra $|w - a|$ bus lines and the next $s = |t|$ inputs are for encoder. There are $|w|$ outputs in total, which come from the extra bus mixed with the MERGE network outputs driven by the encoder.

Extra toppings

If we were to assemble the block $\mathbf{D}_s \langle I_1 s, I_3(w, t) \rangle$ containing the SHUNT cascade $I_1 s$ and the MERGE network $I_3(w, t)$ for a word $w \in c$ and a list $t = (I_0 c) w$ of words in c intersecting w , it would have

the first $s = |t|$ outputs from the TOGGLE array in the SHUNT cascade connected to the s inputs on the encoder in the MERGE network as it should, bringing together most of the slice pertaining to the word w , but it would still need to be topped off with a JOIN in the front and a FORK in the back. There is also a back end JOIN visible in Figure 13.7 for each slice, which we defer for the moment. Attaching a multi-way JOIN to the MERGE network is straightforward in an expression

$$\mathbf{L}_{|w|} \langle \mathbf{D}_s \langle I_1 s, I_3(w, t) \rangle, \mathbf{JOIN} |w| \rangle$$

for a block with the JOIN output last and the first input to the first SHUNT first, so that a roll and one more connection from the JOIN to the SHUNT

$$\mathbf{Z}(\mathbf{L}_{|w|} \langle \mathbf{D}_s \langle I_1 s, I_3(w, t) \rangle, \mathbf{JOIN} |w| \rangle \uparrow 1)$$

finishes everything but the FORK. The FORK is driven by the last and only remaining exposed output from the SHUNT cascade, which has become the first output from the block above, and the FORK output arity not coincidentally is also s , because it is responsible for broadcasting a reset signal to every slice represented in t . A block like

$$\mathbf{ZR}(\mathbf{Z}(\mathbf{L}_{|w|} \langle \mathbf{D}_s \langle I_1 s, I_3(w, t) \rangle, \mathbf{JOIN} |w| \rangle \uparrow 1), \mathbf{FORK} s)$$

would conclude the construction with s reset inputs followed by $|w|$ MERGE network inputs, and s exposed TOGGLE outputs followed by s FORK outputs. The hitherto unused TOGGLE outputs are destined for the back end JOIN network yet to be considered as shown in Figure 13.7. The whole slice except for the back end JOIN therefore can be abbreviated hereafter as $I_4(c, w)$ in terms of a function

$$I_4 = \lambda c. (\lambda t. (\lambda s. \mathbf{ZR}(\mathbf{Z}(\mathbf{L}_{|w|} \langle \mathbf{D}_s \langle I_1 s, I_3(w, t) \rangle, \mathbf{JOIN} |w| \rangle \uparrow 1), \mathbf{FORK} s)) |t|) (I_0 c) w.$$

Stacking the slices

The decoder requires one slice of the form $I_4(c, w)$ for each code word $w \in c$, with the reset outputs from each slice connected to the reset inputs of some subset of other slices. These connections can be specified next.

The number of reset inputs and outputs on each slice is not constant but varies with the word w in terms used above as $s = |t| = |(I_0 c) w|$, so a simple approach to separating the reset terminals from the rest of the terminals on the slices is to roll them off individually in a block $I_5 c$ given by a function

$$I_5 = \lambda c. (\mathcal{F}_{Z1} \lambda(w, z). (\lambda s. \mathbf{R}(I_4(c, w) \downarrow s, z) \uparrow s) |(I_0 c) w|) c^{\circ-1}$$

defined as a fold over the lexicographically ordered list $c^{\circ-1}$ of code words. With the reset inputs and outputs on each slice rolled to the bottom of the parallel combination of itself with its successors in the list, the main effect on the result is to have all of the MERGE network inputs to all slices preceding all of the reset inputs, and all of the TOGGLE outputs from all slices preceding all of the reset outputs, along with a few other effects that are important to note.

- The first inputs to this block form $|c|$ buses, one for each slice, with the j -th bus connected to the MERGE network on the lexicographically j -th slice as one might expect.
- The rest of the inputs form $|c|$ additional buses, but the j -th of these buses connects to the remaining exposed SHUNT inputs (*i.e.*, the so called reset inputs) on the j -th slice in reverse lexicographic order. That is, the last input bus leads to the first slice.

- Within each bus of either group, the order of the lines matters, with each line required to be driven by a specific source as detailed shortly. The order of the lines within each bus of the latter group relative to other lines on the same bus is *not* reversed.
- The organization of the output side is analogous, with $|c|$ groups of TOGGLE outputs followed by $|c|$ groups of FORK outputs (the so called reset outputs). The first group of outputs is in order of the slices but in the second, the last group is associated with the first slice.

To connect the FORK outputs to the SHUNT inputs on the block $I_5 c$ of slices, we restrict attention to these terminals for the moment and disregard the MERGE and TOGGLE terminals. The FORK outputs and SHUNT inputs are each divided into $|c|$ groups such that the j -th group pertains to the slice induced by the word $w = d_j$, where

$$d = r c^{\circ-1}$$

is the list of code words in reverse lexicographic order as explained above and $r = \mathcal{F}_\epsilon \lambda(h, z). z \parallel \langle h \rangle$ is a list reversal function improvised for this occasion. The output terminals on the slice associated with the word w should be connected to input terminals on slices associated with words in the list $i w$ for $i = I_0 c$. The range of terminals on the slice due to a typical word v in this reversed list of buses begins at an offset

$$|\overset{\circ}{b}(d \mid d^{-1} v)|$$

with $(i v)^{-1} w$ being the specific position within this range allotted to the connection from the slice induced by w . These conditions suffice to determine a permutation $(I_6 I_0) d$ according to a function

$$I_6 : (\mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow (\mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})^*)) \rightarrow (\mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{N}^*)$$

defined by

$$I_6 = \lambda i. \lambda d. \overset{\circ}{b}(\lambda w. (\lambda v. |\overset{\circ}{b}(d \mid d^{-1} v)| + (i v)^{-1} w)^* i w)^* d$$

that could be used to specify a permutation network interfacing one group of terminals to the other if they were on separate blocks.

The groups of terminals are actually both on the same block $I_5 c$ in the last positions, of course, requiring something more along the lines of a feedback path

$$\mathbf{Z}^{|p|}(I_5 c \times p \mid |p|)$$

whereby the FORK outputs are first permuted by $p = r (I_6 i) d$ and then rolled from the bottom to the top in preparation for being connected to the SHUNT inputs at the bottom. A further reversal of the permutation by r as shown is needed to preempt the effect of $\mathbf{Z}^{|p|}$ reversing the order of the connections. Ordinarily it would also be necessary to invert a permutation $(I_6 i) d$ if it has been derived as a map from sources to destinations and is repurposed as an output permutation (which maps destinations to sources), but not here because $(I_6 i) d$ is its own inverse. In any case, let the block overall be denoted $I_7 c$ in terms of a function

$$I_7 = \lambda c. (\lambda p. \mathbf{Z}^{|p|}(I_5 c \times p \mid |p|)) (\lambda r. r (I_6 I_0 c) r c^{\circ-1}) \mathcal{F}_\epsilon \lambda(h, z). z \parallel \langle h \rangle.$$

Combining form

The block $I_7 c$ includes everything needed for a decoder of a code c except the front FORK network and the back JOIN network. To specify either of these, we envision the slices numbered from 0 to $|c| - 1$ and seek the destination or source slice numbers of each FORK or JOIN. With regard to the front end, there should be one multi-way FORK for each symbol $a \in \bigcup c$ in the alphabet of the code c , where the a -th FORK connects to each slice whose word has an a in it, which would be any member of

$$c - \mathcal{P}(\mathbb{N} - \{a\}) \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$$

implying a set of slice numbers for the a -th FORK

$$(\mu c^\circ) (c - \mathcal{P}(\mathbb{N} - \{a\})) \in \mathcal{P}(\mathbb{N})$$

and a list of these sets in alphabetical order

$$i = (\mu c^\circ)^* (\lambda a. (c - \mathcal{P}(\mathbb{N} - \{a\}))) (\bigcup c)^{\circ-1} \in \mathcal{P}(\mathbb{N})^*$$

whereby the whole front end FORK network could be expressed

$$(\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK } |f|)^* i$$

and connected to the slices by a permutation network as in

$$(\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK } |f|)^* i \xrightarrow{A_1 i} I_7 c$$

using the permutation $A_1 i$ already defined by Equation 13.4 because it fits the same pattern as the front end FORK network in a basic encoder with a list i of sets of MERGE network ordinals. As for the back end, the TOGGLE outputs from a slice associated with a word w must be connected to the back multi-way JOIN in any other slice whose word intersects w , which would be those whose words are in the set

$$c - \mathcal{P}(\mathbb{N} - w) - \{w\} \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$$

implying the analogous list of sets of slice numbers

$$o = (\mu c^\circ)^* (\lambda w. c - \mathcal{P}(\mathbb{N} - w) - \{w\})^* c^{\circ-1} \in \mathcal{P}(\mathbb{N})^*$$

for a whole back end JOIN* $A_0 o$ by Equation 13.3 based on $(A_0 o)_j$ being defined as the number of terms in o containing j . Connecting the slices to the back end is possible by a permutation network using a permutation $A_1 o$ by Equation 13.4 as in

$$(\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK } |f|)^* i \xrightarrow{A_1 i} I_7 c \xrightarrow{A_1 o} (\mathcal{F} \mathbf{R}) \text{JOIN}^* A_0 o$$

because the interface between the TOGGLE outputs and the JOIN inputs also fits a basic encoder pattern. An expression for $\langle i, o \rangle = I_8 c$ follows from a definition of $I_8 : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathbb{N})^{*2}$ as

$$I_8 = \lambda c. (\mu c^\circ)^{**} \langle (\lambda a. c - \mathcal{P}(\mathbb{N} - \{a\}))^* (\bigcup c)^{\circ-1}, (\lambda w. c - \mathcal{P}(\mathbb{N} - w) - \{w\})^* c^{\circ-1} \rangle$$

allowing the basic decoder combining form $\Omega_i : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$ to be defined as follows.

$$\Omega_i(c) = (\lambda \langle i, o \rangle. (\mathcal{F} \mathbf{R}) (\lambda f. \text{FORK } |f|)^* i \xrightarrow{A_1 i} I_7 c \xrightarrow{A_1 o} (\mathcal{F} \mathbf{R}) \text{JOIN}^* A_0 o) I_8 c$$

This result enables the simplest decoder generating function $\text{Dc} : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$ to be defined as

$$\text{Dc}(c) = \Omega_i c \tag{13.9}$$

by readers who have had enough of decoders already, but more efficient decoders discussed in the rest of this section may interest the more intrepid.

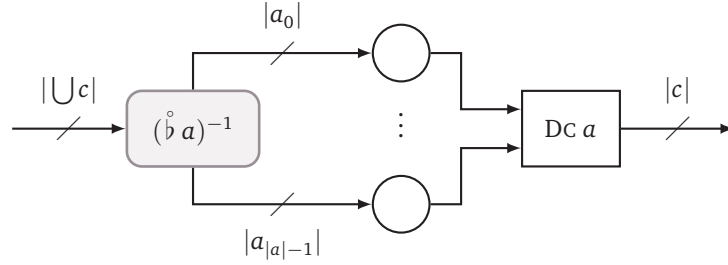


Figure 13.8: a decoder $\Omega_j(c, DC a)$ for a joinable code c with decomposition $a = \mathcal{U}_j c$

13.3.2 Joinable

An improvement on the basic decoder analogous to the front optimized encoder is applicable whenever two symbols always appear together in any code word where either appears, and never separately. In that case, the input lines associated with the two symbols can be fed into a JOIN, and the output of the JOIN fed to the decoder of a reduced code having one less symbol in its alphabet.

A decoder of this form is determined by a list $A_3 c \in \mathcal{P}(\mathbb{N})^*$ of equivalence classes of members of the alphabet $\bigcup c$ of a code c according to a function $A_3 : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathbb{N})^*$ defined by

$$A_3 = \lambda c. ((\pi \lambda i. c - \mathcal{P}(\mathbb{N} - \{i\})) \bigcup c)^{\circ-1}$$

based on Equation 6.6, which is to say a symbol i is equivalent to any other symbol present in the same set of words as i . A decomposition function $\mathcal{U}_j : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}))$ taking a code c to a possibly reduced code $\mathcal{U}_j c$ is obtained in terms of $A_3 c$ by

$$\mathcal{U}_j = \lambda c. (\lambda a. (\mu \mu (a^{-1} \circ \Psi \bigcup_{s \in \mathcal{R}(a)} s \times \{s\})) c) A_3 c$$

which transforms each symbol in each word in c to its equivalence class ordinal with respect to the code $A_3 c$ by similar reasoning to that of Section 13.2.3.

To complete the construction, a combining form $\Omega_j : \mathcal{P}(\mathcal{P}(\mathbb{N})) \times \mathbb{H} \rightarrow \mathbb{H}$ takes a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ and a decoder $x \in \mathbb{H}$ for the reduced code $\mathcal{U}_j c$ to a decoder $\Omega_j(c, x) \in \mathbb{H}$ of the code c as in Figure 13.8. For a list of input equivalence classes $a = A_3 c$, each JOIN network in the front end

$$(\mathcal{F} \mathbf{R}) (\lambda j. \text{JOIN } |j|)^* a$$

has an input arity matching the cardinality $|j|$ of the corresponding class, and with one for each class, it connects to the inner decoder x by a bus of width $|a|$.

$$\mathbf{C}_{|a|} \langle (\mathcal{F} \mathbf{R}) (\lambda j. \text{JOIN } |j|)^* a, x \rangle$$

An input permutation network given by $(\mathring{b} a)^{-1}$ connects the input lines numbered according to the original code c with their destinations on the front end JOIN network, for a result overall as shown.

$$\Omega_j(c, x) = (\lambda a. (\mathring{b} a)^{-1} \times \mathbf{C}_{|a|} \langle (\mathcal{F} \mathbf{R}) (\lambda j. \text{JOIN } |j|)^* a, x \rangle) A_3 c$$

Together the decomposition function \mathcal{U}_j and combining form Ω_j feature in a recursively defined generalization of Equation 13.9 to be discussed in Section 13.3.5.

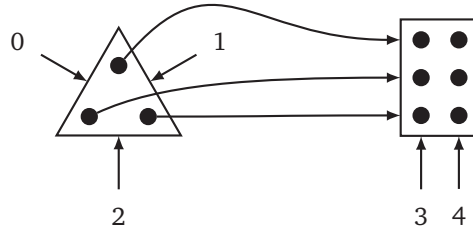


Figure 13.9: A decoder for $\{\{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}, \{0, 1, 4\}, \{0, 2, 4\}, \{1, 2, 4\}\}$ incorporates a decision wait.

13.3.3 Factorable

Other improvements on the basic decoder than the one in Section 13.3.2 may be applicable if we pursue them diligently. For example, if a code c had an alphabet $\{0, 1, 2, 3\}$ and code words

$$c = \{\{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}\} \quad (13.10)$$

then a decoder $\Omega_i c$ would resemble Figure 13.7 with four slices instead of three and would work perfectly well, but a moment's reflection prompts the realization that a 2-by-2 decision wait would do the same job at a lower cost. If the code did not include the word $\{1, 3\}$, then an even better solution would be available in the way of an LJOIN (Figure 9.13).

The previous examples may seem contrived because a decision wait probably would have been a more obvious solution than a decoder in any practical applications giving rise to them, but sometimes an optimization exploiting this relationship might not be so obvious. For example, the code

$$c = \{\{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}, \{0, 1, 4\}, \{0, 2, 4\}, \{1, 2, 4\}\}$$

can not be decoded by a decision wait alone, but can be decoded by a combination of a TRIA and a 3-by-2 decision wait as shown in Figure 13.9. It boggles the mind to contemplate spotting such a pattern when it involves higher dimensional or sparse decision waits. What can be done to ensure never missing this trick?

An awkward way of approaching this question is to reframe it as the question of whether there are two antichains $f, g \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ with disjoint alphabets such that every word $w \in c$ in a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ is expressible as a union $w = u \cup v$ for some $(u, v) \in f \times g$. If so, then c is a **factorable code** with factors f and g . The problem of implementing the decoder would then reduce to that of implementing a separate decoder for each factor, which can be done because the factors are antichains, and feeding the outputs from these decoders into a decision wait whose dimensionality matches the number of factors (e.g., a planar decision wait if there are two factors), with one decoder output bus connected to each dimensional axis on the decision wait.

A less awkward way of approaching this question, which occupies the rest of this section, starts with a decomposition function

$$\mathcal{U}_f : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}))^*$$

taking a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ to a list $\mathcal{U}_f c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))^*$ of its factors, however many factors there are, preferably obtained more easily than by an exhaustive search. Following that, a combining form

$$\Omega_f : \mathcal{P}(\mathcal{P}(\mathbb{N})) \times \mathbb{H}^* \rightarrow \mathbb{H}$$

taking a factorable code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ and a list $x \in \mathbb{H}^*$ of the decoders of its factors to a decoder $\Omega_f(c, x) \in \mathbb{H}$ for c conjures up the implied multidimensional sparse decision wait and permutation networks necessary to combine the individual decoders x . Together these two functions determine yet another case in a recurrence defining a general decoder to come in [Section 13.3.5](#).

Decomposition function

To follow up specifically on the matter of a decomposition function, any factor f of a code c is fully determined by its alphabet $s = \bigcup f$, because f can be reconstructed from just the alphabet and the code as $f = (\mu \lambda w. w \cap s) c$ given that all factors must have mutually disjoint alphabets. Factoring a code c is therefore equivalent to finding the right partition on its alphabet $\bigcup c$. Furthermore, any code c has at least one factor, namely itself, so there is always a starting point for finding more. If we know that $a \subset \mathcal{P}(\bigcup c)$ is a set of alphabets of the factors of a code, the next step should be to inspect each alphabet $s \in a$ to ascertain whether the corresponding factor can be decomposed further. Naively we could test all $2^{|s|}$ subsets of s for being the alphabet of a factor, but if removing any one symbol from s disqualifies it, there is no need to look further, so a restriction to the $|s|$ subsets $u \subset s$ with $|u| = |s| - 1$ is enough for one step. These subsets u are expressible formally as members of the set

$$t = (\mu \lambda i. s - \{i\})s.$$

To be the alphabet of a factor, u must intersect every word $w \in c$ so we write

$$v = \{u \in t \mid \forall w \in c. u \cap w \neq \emptyset\} = \bigcup_{u \in (\mu \lambda i. s - \{i\})s} (\lambda k. \langle \emptyset, \{u\} \rangle_k) \delta_{\emptyset}^{c \cap \mathcal{P}(\mathbb{N}-u)}$$

for the set of all alphabets of smaller factors than that of s inferred from t . If v is empty, then the alphabet s should be retained as representing a factor that can not be decomposed further, but otherwise we should disregard it henceforth and examine the expanded set of alphabets

$$\bigcup_{s \in a} (\lambda v. \langle v, \{s\} \rangle_{\delta_{\emptyset}^v}) \bigcup_{u \in (\mu \lambda i. s - \{i\})s} (\lambda k. \langle \emptyset, \{u\} \rangle_k) \delta_{\emptyset}^{c \cap \mathcal{P}(\mathbb{N}-u)}$$

at our next step. Let $A_4 c \in \mathcal{P}(\mathcal{P}(\bigcup c))$ denote the set of all smallest alphabets of factors of c thus obtained in any number of steps with $A_4 : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}))$ defined by

$$A_4 = \lambda c. (\lambda a. \bigcup_{s \in a} (\lambda v. \langle v, \{s\} \rangle_{\delta_{\emptyset}^v}) \bigcup_{u \in (\mu \lambda i. s - \{i\})s} (\lambda k. \langle \emptyset, \{u\} \rangle_k) \delta_{\emptyset}^{c \cap \mathcal{P}(\mathbb{N}-u)})^{\infty} \{\bigcup c\}.$$

Obtaining the set $a = A_4 c$ of alphabets s as above allows some efficiencies in exchange for an inexact solution insofar as the alphabets s are not guaranteed disjoint. To compensate, an alphabet $s \in a$ extended to encompass all other members of a that intersect it as well as any that intersect those is obtained partly by the percolation

$$(\rho \lambda r. a - \mathcal{P}(\mathbb{N} - r)) \{s\}$$

based on [Equation 6.4](#) and merged into the single alphabet

$$s' = (\lambda p. \bigcup p) (\rho \lambda r. a - \mathcal{P}(\mathbb{N} - r)) \{s\}$$

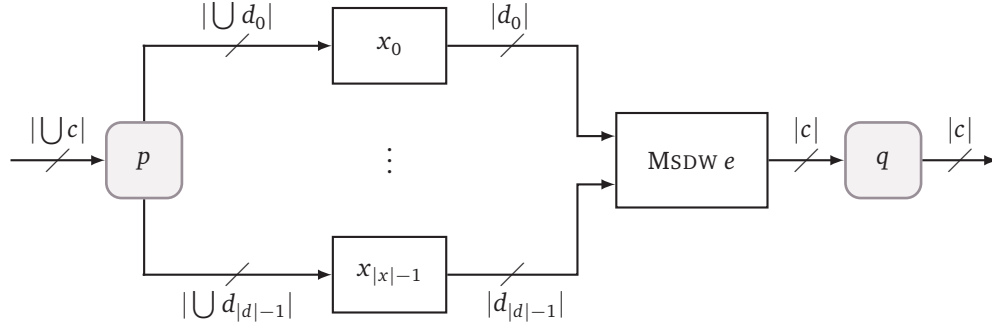


Figure 13.10: A decomposition $d = \mathcal{U}_f c$ of a factorable code c determines an input permutation network $p = (\overset{\circ}{b} (\lambda f. \bigcup f)^* d)^{-1}$, decision wait coordinates $e = (\mu \lambda w. (\lambda i. d_i^\circ (w \cap \bigcup d_i))^* \iota_{|d|}) c$, and an output permutation $q = (\lambda b. c^\circ \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i)^* e^{\circ-1}$ in the decoder $\Omega_f(c, x)$.

suggesting an expression for a set of mutually disjoint alphabets derived from a as

$$(\mu \lambda s. (\lambda p. \bigcup p) (\rho \lambda r. a - \mathcal{P}(\mathbb{N} - r)) \{s\}) a.$$

This set is suitable for inferring the factors of c as described above by $(A_5 c) A_4 c$ with

$$A_5 : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow (\mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N}))))$$

defined by

$$A_5 = \lambda c. \lambda a. (\mu \lambda s'. (\mu \lambda w. w \cap s') c) (\mu \lambda s. (\lambda p. \bigcup p) (\rho \lambda r. a - \mathcal{P}(\mathbb{N} - r)) \{s\}) a.$$

The set $f = (A_5 c) A_4 c$ of factors of c always satisfies $\bigcup \bigcup f = \bigcup c$ as it must if c is an antichain, meaning that no symbols in the alphabet of the code are omitted. In lieu of a proof of this proposition presumably obliging a formally verified implementation of the algorithm to go with it, a decomposition function taking the easy way out incorporates a sanity check

$$\mathcal{U}_f = \lambda c. (\lambda f. (\lambda a. (\lambda i. \langle \epsilon, f^{\circ-1} \rangle_i) \delta_{\bigcup_c^a} \bigcup f) (A_5 c) A_4 c \quad (13.11)$$

to yield the lexicographically ordered list of factors only if it is true.

Combining form

The generalization of Figure 13.9 to any factorable code c includes any number of decoders $x \in \mathbb{H}^*$ connected in parallel to a multidimensional sparse decision wait as shown in Figure 13.10, along with input and output permutation networks to make the whole combination behaviorally equivalent to a decoder of c . For this construction to be described in terms of the combining form $\Omega_f(c, x)$ mentioned previously, each term x_i must be a decoder for the i -th factor d_i where $d = \mathcal{U}_f c$ is the decomposition defined by Equation 13.11.

Input permutation network It is easy to dispense with the input permutation network first by noting that the decoder x_i of the i -th factor d_i is allocated a subset $s = \bigcup d_i \subset \bigcup c$ of the externally visible inputs for $0 \leq i < |d|$. A list of the input alphabet symbols in $\bigcup c$ in order of the decoders receiving them and in numerical order relative to their neighbors on the same decoder

$$\overset{\circ}{b} (\lambda f. \bigcup f)^* d$$

specifies the externally visible bus line to be connected to each terminal in the order they appear internally, hence the inverse of the permutation describing the input permutation network. Along with the array of decoders $(\mathcal{F} \mathbf{R}) x$, we have enough to make a start at defining the combining form

$$\Omega_f(c, x) = (\lambda d. (\overset{\circ}{b} (\lambda f. \bigcup f)^* d)^{-1} \times \mathbf{C}_{|\overset{\circ}{b}|} \langle (\mathcal{F} \mathbf{R}) x, \ddot{\Omega}_f(c, d) \rangle \mathcal{U}_f c \quad (13.12)$$

in terms of a block $\ddot{\Omega}_f(c, d) \in \mathbb{H}$ yet to be determined that includes the sparse decision wait and the output permutation network.

Sparse decision wait With regard to the sparse decision wait, the requirement for the decoder to emit exactly one signal along a 1-hot channel for any chosen input word $w \in c$ implies a bijective correspondence between members of the code c and members of a set $e \in \mathcal{P}(\mathbb{N}^*)$ specifying the sparse decision wait coordinates in the manner described at length in [Chapter 11](#). This observation combined with the plan of one dimension for each factor suggests sparse decision wait coordinates e based on an expression of some form for a typical point $b \in e$ of length $|b| = |d|$ in terms of a typical word $w \in c$. Furthermore, because the output bus from the i -th decoder is connected to the i -th dimensional axis of the decision wait, b_i must range from 0 to $|d_i| - 1$.

If these conditions do not yet give it away, the same intuition applies to a coordinate b_i of a point $b \in e$ as in the root block of a dendriform sparse decision wait with the decoders x taking the place of the leaf blocks. That is, its value should match the number of the output terminal that emits a signal on the i -th decoder whenever the word w associated with b is received on the main input bus. Following from the discussion of the input permutation network, the i -th decoder receives only the subset

$$w \cap \bigcup d_i$$

of symbols in the word w , and therefore emits a signal on the output terminal numbered

$$b_i = d_i^\circ (w \cap \bigcup d_i) \quad (13.13)$$

whereby the whole point $b \in e$ follows immediately as

$$b = (\lambda i. d_i^\circ (w \cap \bigcup d_i))^* \iota_{|d|}$$

and the complete sparse decision wait specification as

$$e = (\mu \lambda w. (\lambda i. d_i^\circ (w \cap \bigcup d_i))^* \iota_{|d|}) c.$$

Output permutation network Arranging for the n -th visible output bus line to transmit a signal whenever the lexicographically n -th input word $w \in c$ is received as an input generally requires an output permutation network to untangle it. Looking backwards from the output side, we see that

the m -th output terminal on the sparse decision wait MSDW e emits a signal when the word $w \in c$ corresponding to the lexicographically m -th point

$$b = (e^{\circ-1})_m$$

in e is received as an input. Each coordinate b_i of the point b accounts for a subset of the corresponding word w per Equation 13.13, so putting together the subsets due to all coordinates in the point should reconstruct the whole input word

$$w = \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i$$

whose ordinal relative to the rest of the code words in c

$$n = c^\circ w = (\lambda b. c^\circ \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i) (e^{\circ-1})_m$$

is the position where we would like the signal to appear on the output bus, not the m -th where we actually see it. A list of these desired positions ordered lexicographically by the points in e and hence by the output terminals on MSDW e

$$(\lambda b. c^\circ \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i)^* e^{\circ-1}$$

determines an output permutation network for the multidimensional sparse decision wait

$$\text{MSDW } e \times ((\lambda b. c^\circ \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i)^* e^{\circ-1})^{-1}$$

that puts the signal acknowledging each word w in the position where it belongs when the list is inverted to map word ordinals n to point ordinals m . Consequently the term $\ddot{\Omega}_f(c, d)$ representing the whole decision wait in Equation 13.12 with the permutation network built in is given by

$$\ddot{\Omega}_f(c, d) = (\lambda e. \text{MSDW } e \times ((\lambda b. c^\circ \bigcup_{i \in \mathcal{D}(b)} d_i^{\circ-1} b_i)^* e^{\circ-1})^{-1}) (\mu \lambda w. (\lambda i. d_i^\circ (w \cap \bigcup d_i))^* \iota_{|d|}) c$$

to complete the definition of the factorable decoder combining form.

13.3.4 Partitionable

A less costly and simpler way of decomposing a code than factoring it is to split it into codes that allow decoding in parallel by multiple decoders with only a front end and a back end permutation network to combine them (*i.e.*, with no need for a decision wait). This decomposition applies to any **partitionable code**, which is any code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ expressible as the union $c = d \cup e$ of disjoint non-empty subsets d and e with disjoint alphabets. Rare but not unknown, a 1-hot code is a ubiquitous example of a partitionable code.



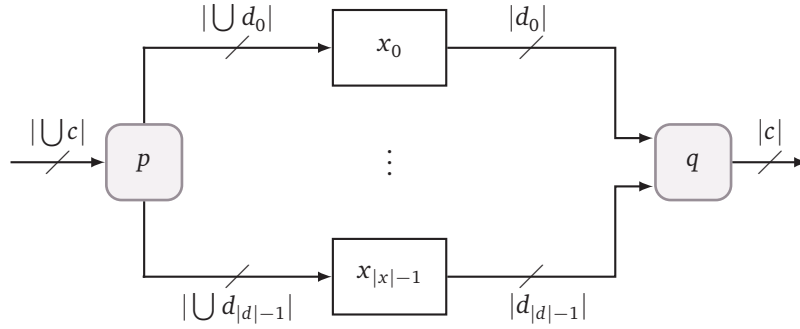


Figure 13.11: A partitionable combination $\Omega_k(c, x)$ determined by a decomposition $d = \mathcal{U}_k c$ has an input permutation $p = (\overset{\circ}{\nu} (\lambda f. \bigcup f)^* d)^{-1}$ and an output permutation $q = (c^{\circ*} \overset{\circ}{\nu} d)^{-1}$.

Decoding a partitionable code c by an ensemble of basic decoders offers no advantage over decoding the whole code by a single basic decoder $\Omega_i c$. However, some advantage may accrue if some subset is factorable or joinable and decoded accordingly. It is worthwhile for this reason to seek a decomposition function and a corresponding combining form for partitionable codes. That is, partitioning them may enable other optimizations in the context of a hierarchically constructed decoder as in Section 13.3.5.

Decomposition function

A decomposition function $\mathcal{U}_k : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}))^*$ needs to take a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ to a list of mutually disjoint subsets $\mathcal{U}_k c \in \mathcal{P}(c)^*$ with mutually disjoint alphabets such that $|\mathcal{U}_k c|$ is greater than 1 whenever the code c is partitionable. A lazy way to express the function is to envision *growing* an equivalence class from each word $w \in c$ by percolating it outward to anything it intersects until it can grow no further because there are no other intersecting words.

$$(\rho \lambda v. c - \mathcal{P}(\mathbb{N} - v)) \{w\} \in \mathcal{P}(c)$$

Operating similarly on every word $w \in c$ yields the full partition

$$(\mu \lambda w. (\rho \lambda v. c - \mathcal{P}(\mathbb{N} - v)) \{w\}) c \in \mathcal{P}(\mathcal{P}(c))$$

with any distinct words that belong together growing into the same class. Disregarding the obviously more efficient procedure for computing this result that springs to mind (for no other reason than brevity in exposition), we capture the lexicographically ordered list of classes formally as follows.

$$\mathcal{U}_k = \lambda c. ((\mu \lambda w. (\rho \lambda v. c - \mathcal{P}(\mathbb{N} - v)) \{w\}) c)^{\circ-1} \tag{13.14}$$

Combining form

The method for combining the decoders $x \in \mathbb{H}^*$ of the subsets $\mathcal{U}_k c \in \mathcal{P}(c)^*$ of a partitionable code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ into a decoder $\Omega_k(c, x) \in \mathbb{H}$ for the whole code c is depicted in Figure 13.11. As a parallel combination $(\mathcal{F} \mathbb{R}) x$ of the given decoders with permutation networks, it incurs no additional cost.

The input permutation network is completely analogous to that of a factorable combination because the input alphabets of the subsets are mutually disjoint. Based on the discussion of Equation 13.12, we can write

$$(\overset{\circ}{b} (\lambda f. \cup f)^* d)^{-1}$$

for the input permutation induced by a decomposition $d = \mathcal{U}_k c$ without further comment.

As for the output permutation, it must route the sequence of output terminals on the array of decoders $(\varepsilon \mathbf{R}) x$, whose m -th terminal transmits a signal when the code word

$$w = (\overset{\circ}{b} d)_m$$

is received, to an externally visible output bus whose n -th line transmits a signal when w is received, where $n = c^\circ w$ is the lexicographic ordinal of w , for any word $w \in c$. The list

$$c^{\circ*} \overset{\circ}{b} d$$

enumerates the lexicographic ordinals n of all words $w \in c$ ordered by the terminal numbers m acknowledging them, so its inverse maps the desired observable bus line number n to its source terminal number m on the array and makes it useful as an output permutation.

These two permutations suffice to determine the combining form $\Omega_k : \mathcal{P}(\mathcal{P}(\mathbb{N})) \times \mathbb{H}^* \rightarrow \mathbb{H}$ for partitionable codes as follows.

$$\Omega_k(c, x) = (\lambda d. (\overset{\circ}{b} (\lambda f. \cup f)^* d)^{-1} \times (\varepsilon \mathbf{R}) x \times (c^{\circ*} \overset{\circ}{b} d)^{-1}) \mathcal{U}_k c \quad (13.15)$$

13.3.5 General

A smarter alternative decoder generating function to that of Equation 13.9 taking advantage of the optimizations developed subsequently in Section 13.3.2 through Section 13.3.4 is now at hand. With the help of a local renumbering function $\dot{\eta} : \mathcal{P}(\mathcal{P}(\mathbb{N}))^* \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}))^*$ defined as

$$\dot{\eta} = (\lambda d. \mu \mu (\cup d)^\circ)^* \quad (13.16)$$

transforming any factor or subset d obtained by Equation 13.11 or Equation 13.14 to a corresponding code satisfying Equation 13.1 (cf. Equation 11.4) we have the following recurrence.

$$\text{DC}(c) = \begin{cases} \mathbf{ZI} & \text{if } c = \emptyset \\ \Omega_k(c, \text{DC}^* \dot{\eta} \mathcal{U}_k c) & \text{if } |\mathcal{U}_k c| > 1 \\ \Omega_j(c, \text{DC} \mathcal{U}_j c) & \text{if } |\mathcal{U}_k c| = 1 \wedge |\cup \mathcal{U}_j c| < |\cup c| \\ \Omega_f(c, \text{DC}^* \dot{\eta} \mathcal{U}_f c) & \text{if } |\mathcal{U}_k c| = 1 \wedge |\cup \mathcal{U}_j c| = |\cup c| \wedge |\mathcal{U}_f c| > 1 \\ \Omega_i c & \text{otherwise} \end{cases} \quad (13.17)$$

Whether it makes sense to call the empty set $c = \emptyset$ a code, there is no harm designating \mathbf{ZI} as its decoder, the circuit having no inputs or outputs, thus covering a case that would be undefined in the alternative. Nor is there a need for any great variety of decomposition strategies, because the one implicit in this recurrence is generally applicable. That is, the partitionable decomposition is always preferable when it applies because it incurs no cost. The joinable decomposition is the next best choice, followed by the factorable, with the basic decomposition only as a last resort.

13.4 Completion detectors

Another thing we might want to do with a delay insensitive code besides decoding it is to build a **completion detector** for it. Completion detectors for 1-hot codes are mentioned in [Chapter 10](#) and a completion detector for a 2-bit dual rail code is used as an example in [Chapter 3](#), but completion detectors are applicable to any delay insensitive code. A completion detector for a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ is similar to its decoder in that they both have $|\bigcup c|$ inputs, but whereas a decoder has a number of outputs equal to the code size $|c|$, a completion detector has only one output regardless of the code size. An output signal from a completion detector indicates that a code word has been received, but does not identify the code word. Completion detectors are useful in various contexts, but our present interest in them is as a prerequisite to the design of transcoder circuits in [Section 13.5](#).

Based on this description, it may seem trivial to construct a completion detector for any code c by connecting a decoder $\text{DC}(c)$ given by [Equation 13.17](#) to a 1-hot completion detector $\text{MERGE } |c|$ in a cascade $\text{C}_{|c|} \langle \text{DC}(c), \text{MERGE } |c| \rangle$. However, a completion detector in this form is often suboptimal and is appropriate only when there is no better alternative. A completion detector for any constant weight code can be implemented at a lower cost as a majority gate such as the one developed in [Chapter 2](#). Furthermore, even if a code is not a constant weight code, it may have factors or subsets that are, allowing at least partial use of majority gates in its completion detector if we make a point of constructing it hierarchically.



To have efficient completion detectors for arbitrary codes, we need to generalize the 2-of-3 majority gate developed in [Chapter 2](#) to a k -of- n majority gate for arbitrary $k, n \in \mathbb{N}$ with $0 < k \leq n$, but as [Figure 2.10](#) shows, the majority gate depends on a sequencer, which must also be generalized to any size. [Section 13.4.1](#) therefore attends to sequencers, followed by [Section 13.4.2](#) on majority gates, before a recursively defined completion detector is proposed in [Section 13.4.3](#).

13.4.1 Sequencers

It is not much of a stretch to upgrade the manually designed 2-way sequencer shown in [Figure 9.11](#) to the n -way version shown in [Figure 13.12](#) based on the intervening development of multi-way arbiters and decision waits. Following [Equation 9.17](#), the central block becomes a parallel combination of n MERGE primitives, and n -way arbiter, and a parallel combination of n TOGGLE primitives cascaded together.

$$\text{C}_n \langle \text{MERGE}^{n \leftarrow 1}_2, \text{ARB } n, \text{TOGGLE}^{n \rightarrow 1}_2 \rangle$$

This block has $2n$ inputs to the MERGE network and $2n$ outputs from the TOGGLE network, each segregated into two buses of width n . The rest of the components are in a block of the form

$$\text{D} \langle \text{PUSH}, \text{DW}(n, 1) \rangle$$

with $n + 1$ inputs and n outputs, whose first input is to the PUSH , and whose last n inputs are to the decision wait rows. To connect the dotted outputs from the TOGGLE network to the decision wait rows in the reverse order, we write

$$\text{Z}^n \text{R} \langle \text{C}_n \langle \text{MERGE}^{n \leftarrow 1}_2, \text{ARB } n, \text{TOGGLE}^{n \rightarrow 1}_2 \rangle, \text{D} \langle \text{PUSH}, \text{DW}(n, 1) \rangle \rangle$$

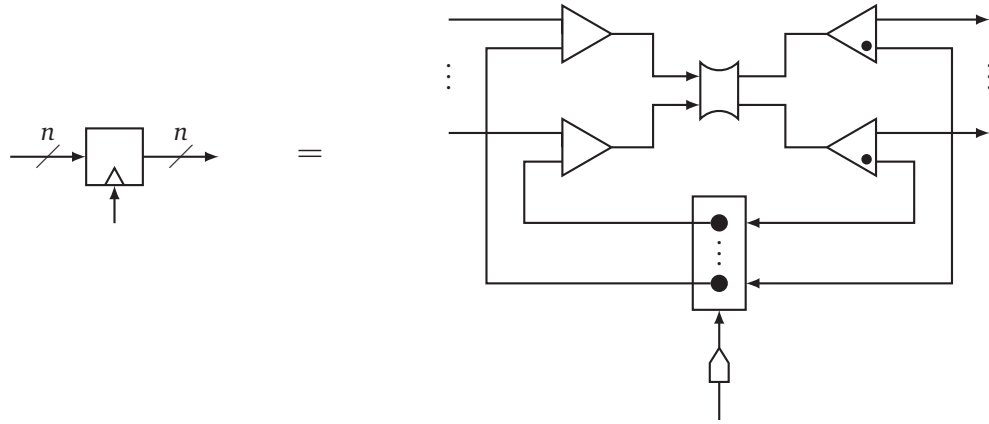


Figure 13.12: An n -way sequencer is made from an n -by-1 decision wait, an n -way arbiter, n MERGE primitives, n TOGGLE primitives, and a PUSH. (cf. Figure 9.11).

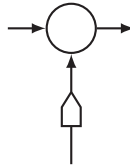


Figure 13.13: a 1-way sequencer, $F\langle \text{PUSH, JOIN} \rangle \uparrow 1$

resulting in a block with $2n + 1$ inputs and $2n$ outputs, where the first $2n$ inputs are to the MERGE network, the remaining input is to the PUSH, the first n outputs are from the TOGGLE network, and the last n outputs are from the decision wait. To roll n of the MERGE inputs to the end and the n decision wait outputs to the beginning, we write

$$(Z^n R(C_n \langle \text{MERGE}^n \leftarrow_2^1, \text{ARB } n, \text{TOGGLE}^n \rightarrow_2^1 \rangle, D \langle \text{PUSH, DW}(n, 1) \rangle)) \uparrow n$$

and then connect the decision wait outputs to n of the MERGE inputs by writing

$$Z^n ((Z^n R(C_n \langle \text{MERGE}^n \leftarrow_2^1, \text{ARB } n, \text{TOGGLE}^n \rightarrow_2^1 \rangle, D \langle \text{PUSH, DW}(n, 1) \rangle)) \uparrow n)$$

where these connections are also in the reverse order to compensate for the connections to the decision wait inputs being in the reverse order. That is, the last decision wait output goes to the first MERGE because the last decision wait column input comes from the first TOGGLE.

We could leave the definition of an n -way sequencer at that, or throw in an optimized version for the degenerate case of $n = 1$ as shown in Figure 13.13, saving a MERGE and a TOGGLE. Taking the latter approach leads to the following definition for the sequencer generating function $\text{SEQ} : \mathbb{N} \rightarrow \mathbb{H}$, which is defined only for positive values of n .

$$\text{SEQ}(n) = \begin{cases} F\langle \text{PUSH, JOIN} \rangle \uparrow 1 & \text{if } n = 1 \\ Z^n ((Z^n R(C_n \langle \text{MERGE}^n \leftarrow_2^1, \text{ARB } n, \text{TOGGLE}^n \rightarrow_2^1 \rangle, D \langle \text{PUSH, DW}(n, 1) \rangle)) \uparrow n) & \text{otherwise} \end{cases}$$

13.4.2 Majority gates

To generalize a majority gate, we can work directly from Figure 2.10 by transcribing it to a block combinator expression. For a k -of- n majority gate, the TOGGLE has to count up to k signals received, so the primitive TOGGLE should be replaced by a network TOGGLE k . The MERGE network following the TOGGLE network therefore needs k inputs, giving it the form MERGE k . Combining the MERGE network with the FORK first, we have

$$\mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle$$

which has the FORK input first followed by $k - 1$ MERGE inputs, and a FORK output followed by a MERGE output. The FORK input should be connected to the last output from the TOGGLE network so that the output from the FORK is transmitted externally only after the TOGGLE network has cycled through the first $k - 1$. To roll the FORK input to the end, we write

$$\mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1$$

so that this block can be combined directly with the TOGGLE network in a cascade with k connections.

$$\mathbf{C}_k \langle \text{TOGGLE } k, \mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1 \rangle$$

On the front end we have a block consisting of a sequencer with n inputs connected necessarily to an n -way MERGE,

$$\mathbf{C}_n \langle \text{SEQ } n, \text{MERGE } n \rangle$$

which is simple to cascade with the back end block just constructed above.

$$\mathbf{C} \langle \mathbf{C}_n \langle \text{SEQ } n, \text{MERGE } n \rangle, \mathbf{C}_k \langle \text{TOGGLE } k, \mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1 \rangle \rangle$$

This block has $n + 1$ inputs, the last being to the acknowledgment on the sequencer, and 2 outputs, the first being from the FORK and the second being from the MERGE network. To connect the MERGE network output to the sequencer, we have to roll the latter to the beginning by writing

$$\mathbf{C} \langle \mathbf{C}_n \langle \text{SEQ } n, \text{MERGE } n \rangle, \mathbf{C}_k \langle \text{TOGGLE } k, \mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1 \rangle \rangle \downarrow 1$$

and then make the connection via the Z combinator.

$$\mathbf{Z} \langle \mathbf{C} \langle \mathbf{C}_n \langle \text{SEQ } n, \text{MERGE } n \rangle, \mathbf{C}_k \langle \text{TOGGLE } k, \mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1 \rangle \rangle \downarrow 1 \rangle$$

This expression suffices to define most majority gates, but two significant optimizations are possible in edge cases. A k -of- n majority gate satisfying $k = n$ reduces to an n -way JOIN, and the case of $k = 1$ reduces to an n -way MERGE. If k and n are both 1, then both the JOIN and the MERGE network reduce to a wire so there is no need to distinguish between them. Putting these ideas together, we have the following definition for a function $\text{MG} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a pair of positive numbers $(k, n) \in \mathbb{N} \times \mathbb{N}$ with $k \leq n$ to a k -of- n majority gate $\text{MG}(k, n) \in \mathbb{H}$.

$$\text{MG}(k, n) = \begin{cases} \langle \text{MERGE } n, \text{JOIN } n \rangle_{\delta_n^k} & \text{if } k \in \{1, n\} \\ \mathbf{Z} \langle \mathbf{C} \langle \mathbf{C}_n \langle \text{SEQ } n, \text{MERGE } n \rangle, \mathbf{C}_k \langle \text{TOGGLE } k, \mathbf{F}\langle \text{FORK}, \text{MERGE } k \rangle \uparrow 1 \rangle \rangle \downarrow 1 \rangle & \text{otherwise} \end{cases}$$

13.4.3 Recurrence

A recurrence defining the completion detector for an arbitrary code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ can be split into four cases for best results. If the code is partitionable, then a separate completion detector for each subset combined by a MERGE network should be the first choice. If the code is not partitionable but is factorable, then a separate completion detector for each factor combined by a JOIN network is preferable. If the code is neither partitionable nor factorable but has a constant weight, then a majority gate should be used, and when all else fails, a decoder cascaded with a MERGE network as proposed at the beginning of Section 13.4 is needed. Note that constant weight codes that are also factorable, such as dual rail codes, are worth factoring first.

The same decomposition functions \mathcal{U}_f and \mathcal{U}_k defined for decoders by Equation 13.11 and Equation 13.14 are applicable to completion detectors, but slightly modified versions of the combining forms Ω_k and Ω_f are appropriate.

$$\begin{aligned}\dot{\Omega}_f(c, x) &= (\mathring{b}(\lambda f. \bigcup f)^* \mathcal{U}_f c)^{-1} \times \mathbf{C}_{|x|} \langle (\mathcal{F} \mathbf{R}) x, \text{JOIN } |x| \rangle \\ \dot{\Omega}_k(c, x) &= (\mathring{b}(\lambda f. \bigcup f)^* \mathcal{U}_k c)^{-1} \times \mathbf{C}_{|x|} \langle (\mathcal{F} \mathbf{R}) x, \text{MERGE } |x| \rangle\end{aligned}$$

Because every completion detector in the formal parameter x is assumed to have only one output, and the resulting completion detector $\dot{\Omega}_k(c, x)$ or $\dot{\Omega}_f(c, x)$ also has only one output, there is no need for any output permutation network, but there is a need for either a MERGE network or a JOIN network to combine them as noted above. The input permutation networks are identical to those of the original combining forms Ω_f and Ω_k as given by Equation 13.11 and Equation 13.14 respectively.

The rest of the construction for a function $\text{CD} : \mathcal{P}(\mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$ taking a code $c \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$ to a completion detector $\text{CD}(c) \in \mathbb{H}$ follows naturally.

$$\text{CD}(c) = \begin{cases} \dot{\Omega}_k(c, \text{CD}^* \mathring{\eta} \mathcal{U}_k c) & \text{if } |\mathcal{U}_k c| > 1 \\ \dot{\Omega}_f(c, \text{CD}^* \mathring{\eta} \mathcal{U}_f c) & \text{if } |\mathcal{U}_k c| = 1 \wedge |\mathcal{U}_f c| > 1 \\ \text{MG}(\min(\mu \lambda w. |w|) c, |\bigcup c|) & \text{if } |\mathcal{U}_k c| = 1 \wedge |\mathcal{U}_f c| = 1 \wedge |(\mu \lambda w. |w|) c| = 1 \\ \langle \mathbf{C}_{|c|} \langle \text{DC}(c), \text{MERGE } |c| \rangle, \mathbf{Z} \rangle_{\delta^{\emptyset}} & \text{otherwise} \end{cases}$$

The renumbering function $\mathring{\eta}$ is defined by Equation 13.16, and the condition $|(\mu \lambda w. |w|) c| = 1$ specifies that the code c has a constant weight of $\min(\mu \lambda w. |w|) c$. The last case provides for an empty code to have an empty completion detector and a brute force solution otherwise.

13.5 Transcoders

Whereas encoders and decoders always interface a 1-hot channel with something, a **transcoder** interfaces two channels carrying arbitrarily chosen delay insensitive codes. Like an encoder, a transcoder may also be envisioned as a read-only memory or look-up table, but instead of being restricted to a 1-hot address bus, it could be queried by a dual rail or k -of- n address bus, for example. Transcoders are the topic of this section, being the next and last step for this chapter.

Designing a transcoder would seem to be a straightforward matter of cascading a decoder with an encoder, but some brief consideration before dismissing it as trivial raises at least a few notable aspects. If a transcoder is designed in the obvious way, then interfacing between two channels even with moderate bus widths could be prohibitive when their code sizes are large, because the code sizes determine the width of the internal 1-hot channel. This problem is serious enough to motivate

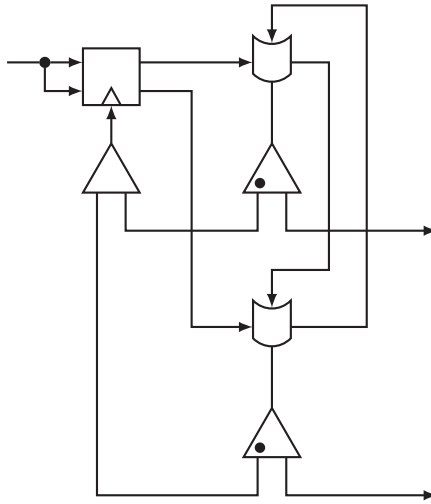


Figure 13.14: A randomizer cell RC acknowledges an input by either of two outputs.

a custom approach to the important special case of conversions between dual rail and Sperner codes in [Chapter 14](#), but some optimizations on space are possible even for the general case. In particular, if some subset of the input code words happens to map to a single output code word, then it may be possible under certain conditions to use a completion detector ([Section 13.4](#)) somewhere in place of a decoder to cut down on the internal bus width.

The last point raises the issue of code sizes not matching exactly. There could be more input than output code words, or more output than input code words, and there could also be repetitions among the output code words similarly to an encoder. An enumeration of the output code words simply as a function of the input words like an encoder specification is not sufficiently expressive for every transcoder that might be of interest. A relation that can be one-to-one, one-to-many, many-to-one, many-to-many, or any combination would be needed to cover everything.

On the other hand, is there ever a good reason for a one-to-many relation as a transcoder specification? Such a specification would allow multiple output code words u , v , and w for a single input code word a , as if the transcoder somehow should choose one at random. Perhaps unexpectedly, as [Figure 13.14](#) shows, there is no impediment in principle to a circuit doing just that, so this interpretation is an option.¹ Rather than arbitrarily restricting the range of expressible specifications, we lose nothing by accepting any that has a valid interpretation and generating the transcoder corresponding to it as faithfully as possible.

Following this plan, an acceptable transcoder specification is a relation

$$t \in \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))$$

whose domain $\mathcal{D}(t)$ represents the input code and whose range $\mathcal{R}(t)$ represents the output code. The relation t is unrestricted except insofar as its domain and range are both expected to be

¹An actual implementation might be partly predictable due to physical asymmetry, crosstalk, or environmental conditions. Do not rely on this circuit as a cryptographically secure source of entropy without careful analysis [[157](#)].

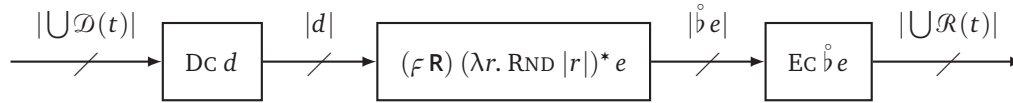


Figure 13.15: A basic transcoder $\Omega_x t$ with a domain $d = \mathcal{D}(t)$ and an encoder specification $e = (\Psi \Pi t)^* d^{\circ-1}$, has a decoder in front, randomizers or a bus in the middle, and an encoder in the back.

antichains satisfying Equation 13.1. As such, it describes a transcoder having exactly $|\bigcup \mathcal{D}(t)|$ input terminals and $|\bigcup \mathcal{R}(t)|$ output terminals. For any pair of words $(a, b) \in t$, a set of signals concurrently received on all input terminals numbered $i \in a$ enables the transcoder to transmit concurrent signals on all output terminals numbered $o \in b$.

The rest of this section develops a method for constructing a transcoder circuit from any specification having this interpretation. Section 13.5.1 presents a basic method for constructing transcoders without any optimizations, Section 13.5.2 proposes a decomposition function and a combining form usable when the input code is partitionable, and Section 13.5.3 describes a transcoder generating function as a recurrence with a view to employing completion detectors where possible as an optimization.

13.5.1 Basic

The transcoder design required when no optimizations are applicable is mainly a decoder cascaded with an encoder as mentioned above, but may have an additional middle stage of non-deterministic circuits as shown in Figure 13.15 sending individual inputs to multiple outputs at random if the specification is a one-to-many relation. If the specification indicates no more than one output for each input code word, then the middle stage should reduce to a bus. We can break it down by stages as follows.

Front end decoder

Whatever comes after it in the other stages, the decoder on the front end must be able to recognize any input code word $a \in d$ in the domain $d = \mathcal{D}(t)$ of the transcoder specification t and emit a signal on the line numbered $n = d^\circ a$. This requirement restricts it to a block of the form DC d by Equation 13.17.

Back end encoder

For the back end encoder stage of the transcoder, we need to infer a specification as a list of code words from the transcoder specification $t \in \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))$, which is a relation. If it were a one-to-one relation, then a lexicographically ordered list $\mathcal{R}(t)^{\circ-1}$ of the range of t would suffice, but in general, any input word $a \in \mathcal{D}(t)$ may be related to multiple output words $b \in \mathcal{R}(t)$. To allow for this possibility, we could consider a slightly more complicated relation

$$\Pi t \in \mathcal{P}(\mathcal{D}(t) \times \mathcal{P}(\mathcal{R}(t)))$$

expressed by the Π operator defined in Equation 6.7, which associates each input code word $a \in \mathcal{D}(t)$ with the set of all output code words $b \in \mathcal{R}(t)$ related to a by t . Technically this relation would be injective for any choice of t , so it could induce a function

$$\Psi \Pi t : \mathcal{D}(t) \rightarrow \mathcal{P}(\mathcal{R}(t))$$

taking any input code word $a \in \mathcal{D}(t)$ to the set of output code words $(\Psi \Pi t) a \in \mathcal{P}(\mathcal{R}(t))$ by Equation 6.1, which could induce a list e of sets of output code words ordered lexicographically by the corresponding input.

$$e = (\Psi \Pi t)^* \mathcal{D}(t)^{\circ-1} \in \mathcal{P}(\mathcal{R}(t))^* \quad (13.18)$$

The resulting list e is not quite valid as an encoder specification because each term of e is a set of code words rather than a specific code word, but a flattened version

$$\overset{\circ}{b} e \in \mathcal{R}(t)^*$$

could specify a back end encoder $\text{Ec } \overset{\circ}{b} e$ handily.

Middle stage randomizers

The back end encoder specification $\text{Ec } \overset{\circ}{b} e$ implies an output arity of $|\overset{\circ}{b} e|$ for the middle stage, while the front end constrains its input arity to $|d| = |\mathcal{D}(t)| = |e|$. To reconcile the output with the input arity, we envision the n -th of $|e|$ outputs from the front end connected to a multi-way generalization of the circuit shown in Figure 13.14 having output arity $|e_n|$. Denoting this circuit $\text{RND } |e_n|$ in terms of a function $\text{RND} : \mathbb{N} \rightarrow \mathbb{H}$ to be defined presently implies a middle stage consisting of an array of the form $(\mathcal{F} \mathbf{R}) (\text{RND } |r|)^* e$, because then the range of $|e_n|$ inputs to the back end associated with the n -th code word are driven directly or indirectly by the n -th output line from the front end via the $|e_n|$ outputs from the n -th block of the middle stage.

We can easily generalize the circuit depicted in Figure 13.14, called a **randomizer** hereafter for lack of a better term, to any arity by building a binary tree, and have it reduce to a wire as required in the case of a single output by writing a recurrence accordingly for $\text{RND} : \mathbb{N} \rightarrow \mathbb{H}$.

$$\text{RND}(m) = \begin{cases} \mathbf{1} & \text{if } m = 1 \\ \mathbf{Z}^2 (\mathcal{F} \mathbf{R}) \text{RC} : \text{RND}^* \langle [m/2], [m/2] \rangle & \text{otherwise} \end{cases} \quad (13.19)$$

The constant $\text{RC} \in \mathbb{H}$ in Equation 13.19 represents the 2-way randomizer, which we can describe by a block combinator expression as follows. A block of two SHUNT and TOGGLE combinations in parallel

$$\mathbf{L} \langle \text{SHUNT}, \text{TOGGLE} \rangle^2$$

has two inputs and three outputs for each combination, the first input of each being the data line in to the SHUNT and the first output being the data line out of the SHUNT (*i.e.*, its horizontal input and output in Figure 13.14). Rolling the first input and the first output of each to the beginning of the block

$$\mathbf{L} \langle \text{SHUNT}, \text{TOGGLE} \rangle^2 \leftarrow_2^1 \rightarrow_3^1$$

enables a connection from the data output of each SHUNT to the control input of the other (*i.e.*, the vertical input in the figure), resulting in a block with two inputs and four outputs.

$$\mathbf{Z}^2 (\mathbf{L} \langle \text{SHUNT}, \text{TOGGLE} \rangle^2 \leftarrow_2^1 \rightarrow_3^1)$$

The four outputs are from the two TOGGLE primitives, whose dotted outputs are rolled to the beginning by the expression

$$(\mathbf{Z}^2(\mathbf{L}\langle\text{SHUNT}, \text{TOGGLE}\rangle^2 \leftarrow_{1_2}^1 \Gamma_3^1)) \Gamma_2^1$$

and connected to a MERGE by two lines in a cascade.

$$\mathbf{F}_2\langle(\mathbf{Z}^2(\mathbf{L}\langle\text{SHUNT}, \text{TOGGLE}\rangle^2 \leftarrow_{1_2}^1 \Gamma_3^1)) \Gamma_2^1, \text{MERGE}\rangle$$

The two remaining inputs are the SHUNT data lines, which can be connected to a FORK and sequencer combination (Section 13.4.1)

$$\mathbf{F}_2\langle\text{FORK}, \text{SEQ } 2\rangle$$

by another two lines in a cascade.

$$\mathbf{Z}\langle\mathbf{F}_2\langle\text{FORK}, \text{SEQ } 2\rangle, \mathbf{F}_2\langle(\mathbf{Z}^2(\mathbf{L}\langle\text{SHUNT}, \text{TOGGLE}\rangle^2 \leftarrow_{1_2}^1 \Gamma_3^1)) \Gamma_2^1, \text{MERGE}\rangle \downarrow 1\rangle$$

This expression leaves two inputs (one to the FORK and one to the sequencer) and three outputs (one from each TOGGLE and one from the MERGE). Rolling the MERGE output to the top enables a connection from it to the sequencer input, which is already at the bottom,

$$\mathbf{Z}\langle\mathbf{C}_2\langle\mathbf{F}_2\langle\text{FORK}, \text{SEQ } 2\rangle, \mathbf{F}_2\langle(\mathbf{Z}^2(\mathbf{L}\langle\text{SHUNT}, \text{TOGGLE}\rangle^2 \leftarrow_{1_2}^1 \Gamma_3^1)) \Gamma_2^1, \text{MERGE}\rangle \downarrow 1\rangle$$

hence sufficient for the 2-way randomizer RC needed in Equation 13.19.

$$\text{RC} = \mathbf{Z}\langle\mathbf{C}_2\langle\mathbf{F}_2\langle\text{FORK}, \text{SEQ } 2\rangle, \mathbf{F}_2\langle(\mathbf{Z}^2(\mathbf{L}\langle\text{SHUNT}, \text{TOGGLE}\rangle^2 \leftarrow_{1_2}^1 \Gamma_3^1)) \Gamma_2^1, \text{MERGE}\rangle \downarrow 1\rangle$$

Combining form

The back end encoder $\text{EC } \overset{\circ}{b} e$ considered previously allows for a cascade with the front and middle stages discussed above

$$\mathbf{C}_{|d|}\langle\text{DC } d, \mathbf{C}_{|\overset{\circ}{b} e|}\langle(\mathcal{F} \mathbf{R}) (\lambda r. \text{RND } |r|)^* e, \text{EC } \overset{\circ}{b} e\rangle\rangle$$

because the arities of the three stages now match up. This expression suggests the following definition for a basic transcoder combining form $\Omega_x : \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$.

$$\Omega_x = \lambda t. (\lambda d. (\lambda e. \mathbf{C}_{|\overset{\circ}{b} e|}\langle\mathbf{C}_{|d|}\langle\text{DC } d, (\mathcal{F} \mathbf{R}) (\lambda r. \text{RND } |r|)^* e\rangle, \text{EC } \overset{\circ}{b} e\rangle) (\Psi \Pi t)^* d^{\circ-1}) \mathcal{D}(t) \quad (13.20)$$

13.5.2 Partitionable

A decomposition of a transcoder into smaller transcoders raises the possibility of locally optimized components even if the decomposition is not advantageous in itself. A decomposition and complementary combining form described briefly in this section work together to this end in a recursively defined transcoder generating function following in Section 13.5.3.

Decomposition function

The decomposition function \mathcal{U}_k for partitionable decoders defined by Equation 13.14 has a direct analog

$$\mathcal{U}_y : \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})) \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))^*$$

for transcoders given by

$$\mathcal{U}_y = \lambda t. (\lambda p. \bigcup_{w \in p} \{w\} \times (\Psi \Pi t) w)^* \mathcal{U}_k \mathcal{D}(t) \quad (13.21)$$

based on the partitionable decomposition $\mathcal{U}_k \mathcal{D}(t)$ of the input code $\mathcal{D}(t)$ for the transcoder specified by t into a list of equivalence classes. Each word $w \in p$ of each equivalence class $p \in \mathcal{R}(\mathcal{U}_k \mathcal{D}(t))$ is also an input word in $\mathcal{D}(t)$, hence associated with the set of output words $(\Psi \Pi t) w \subseteq \mathcal{R}(t)$ and the subset $\{w\} \times (\Psi \Pi t) w \subseteq t$ of the transcoder specification, along with the larger subset obtained by letting w range over p (cf. Equation 13.18). Each of the latter subsets is treated as one of $|\mathcal{U}_y t|$ relations able to specify a separate transcoder. Although the domains of these relations are mutually disjoint by construction, their ranges need not be, a matter to which we return shortly.

Combining form

A combining form $\Omega_y : \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})) \times \mathbb{H}^* \rightarrow \mathbb{H}$ complementary to the decomposition function \mathcal{U}_y proposed above would take a transcoder specification $t \in \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))$ and a list of transcoders $x \in \mathbb{H}^*$ to a transcoder $\Omega_y(t, x) \in \mathbb{H}$ described by t whenever each term x_i is described by the relation $(\mathcal{U}_y t)_i$ for $0 \leq i < |x|$. A partial solution inspired by the partitionable decoder combination in Equation 13.15 features an input permutation network and parallel combination

$$(\overset{\circ}{b} d)^{-1} \times (\mathcal{F} \mathbf{R}) x$$

where d is a list of subsets $d_i = \bigcup \mathcal{D}((\mathcal{U}_y t)_i)$ of the input alphabet allocated to blocks x_i respectively for $0 \leq i < |x|$. The analogous reasoning to that of Equation 13.12 applies.

However, a complete solution must take into account the back end of the combined transcoder, for which no similar permutation network based on flattening a list of subsets $r_i = \bigcup \mathcal{R}((\mathcal{U}_y t)_i)$ exists because they are not necessarily mutually disjoint. For this part, we can draw inspiration from the basic encoder combination discussed in Section 13.2.1, with each transcoder x_i playing the role of a FORK network in Figure 13.2 transmitting the subset r_i of the output alphabet. The same reasoning leads inevitably to a back end MERGE network

$$(\mathcal{F} \mathbf{R}) \text{MERGE}^* A_0 r$$

wherein the j -th MERGE has as many inputs as the number $(A_0 r)_j$ of j 's appearing throughout all sets r_i by Equation 13.3. This network is connected to the front end by a permutation network specified by the permutation $A_1 r$ as in

$$(\overset{\circ}{b} d)^{-1} \times (\mathcal{F} \mathbf{R}) x \xrightarrow{A_1 r} (\mathcal{F} \mathbf{R}) \text{MERGE}^* A_0 r$$

by Equation 13.4, because the i -th transcoder x_i needs to connect to the j -th MERGE for each $j \in r_i$. Consequently we may write

$$\Omega_y(t, x) = (\lambda \langle d, r \rangle. (\overset{\circ}{b} d)^{-1} \times (\mathcal{F} \mathbf{R}) x \xrightarrow{A_1 r} (\mathcal{F} \mathbf{R}) \text{MERGE}^* A_0 r) ((\lambda p. \langle \bigcup \mathcal{D}(p), \bigcup \mathcal{R}(p) \rangle)^* \mathcal{U}_y t)^{\top}$$

as the definition of the combining form using the transpose notation explained in Section 11.1.2.

13.5.3 General

The best opportunity to optimize a transcoder with a specification $t \in \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))$ knocks when every input word $a \in \mathcal{D}(t)$ maps to exactly the same output word $b \in \mathcal{R}(t)$, as indicated whenever $|\mathcal{R}(t)| = 1$ holds. In this case, no encoders, decoders or randomizers are needed because a circuit of the form

$$\mathbf{ZR}(\mathbf{CD} \mathcal{D}(t), \mathbf{FORK} | \bigcup \mathcal{R}(t) |)$$

achieves the same result using a completion detector $\mathbf{CD} \mathcal{D}(t)$ connected to a \mathbf{FORK} network with an output arity matching the output alphabet cardinality. Such a specification is unlikely in isolation, but could come more plausibly from a term $p \in \mathcal{R}(\mathcal{U}_y t)$ for some useful specification t decomposable by \mathcal{U}_y (Equation 13.21). Some reduction in cost would then be achievable by implementing each term separately and combining them by Ω_y .

Following through with this idea involves a small technicality. Decomposing the relation t into parts $p \in \mathcal{R}(\mathcal{U}_y t)$ generally invalidates the conditions that the domain $\mathcal{D}(p)$ and the range $\mathcal{R}(p)$ are codes with consecutively numbered alphabets per Equation 13.1 as assumed in the derivation of the basic transcoder combinator Ω_x (Equation 13.20). There may be gaps in the input or output alphabet of any term due to its missing symbols appearing only in other terms. To correct for this effect, let each pair of words $(a, b) \in p$ be reassigned the value

$$((\mu (\bigcup \mathcal{D}(p)))^\circ a, (\mu (\bigcup \mathcal{R}(p)))^\circ b)$$

so that input symbol $i \in a$ and each output symbol $o \in b$ maps to its ordinal relative only to the input or output alphabet local to p . A renumbering function

$$\ddot{\eta} : \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))^* \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))^*$$

achieves this effect on every pair of words $(a, b) \in p$ in every term $p \in \mathcal{R}(\mathcal{U}_y t)$ for $\ddot{\eta} \mathcal{U}_y t$ when defined in the obvious way (cf. Equation 13.16).

$$\ddot{\eta} = (\lambda p. (\mu \lambda(a, b). ((\mu (\bigcup \mathcal{D}(p)))^\circ a, (\mu (\bigcup \mathcal{R}(p)))^\circ b)) p)^*$$

Resolving this issue now clears the way for a definition of $\mathbf{Tc} : \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})) \rightarrow \mathbb{H}$ as a transcoder generating function by way of a recurrence.

$$\mathbf{Tc}(t) = \begin{cases} \Omega_y(t, \mathbf{Tc}^* \ddot{\eta} \mathcal{U}_y t) & \text{if } |\mathcal{U}_y t| > 1 \\ \mathbf{ZR}(\mathbf{CD} \mathcal{D}(t), \mathbf{FORK} | \bigcup \mathcal{R}(t) |) & \text{if } |\mathcal{U}_y t| = 1 \wedge |\mathcal{R}(t)| = 1 \\ \Omega_x t & \text{otherwise} \end{cases} \quad (13.22)$$

As in Equation 13.17, there is no need for a decomposition strategy any more complicated than this one. The decomposition by \mathcal{U}_y is always preferable when applicable because it incurs no additional cost and may be conducive to the optimization by a completion detector in the second case. In less fortunate circumstances, the strategy is to fall back on a basic transcoder of the form $\Omega_x t$ as in Section 13.5.1.

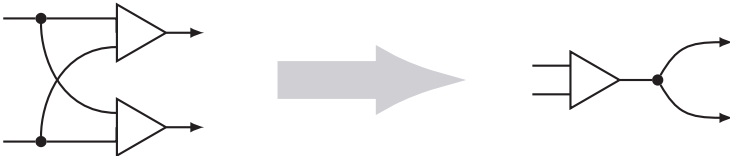


Figure 13.16: a local optimization for encoders

Communication skills

1. What question has this answer?

$$\frac{1}{n} \log_2 \binom{n}{k}$$



2. Simulate the TRIA in [Figure 13.7](#) on paper by tracing the signals with highlighter pens. What is the asymptotic latency or critical path length in general of a k -of- n basic decoder (in terms of k and n)?
3. Develop a formal specification for the low cost alternative basic decoder form mentioned on page [423](#), and compare it to $\Omega_i c$ with regard to cost and performance.
4. Is there a way to build large sequencers as binary trees of small ones, and if so, is it preferable to the method in [Section 13.4.1](#) for either cost or performance?
5. What is the asymptotic average critical path length of a k -of- n majority gate $MG(k, n)$? (An upper bound is sufficient but ignore metastability.) Is there a performance argument for or against using one as a completion detector?
6. What is the optimal open Petri net model of a 2-way randomizer $RND(2)$? (hint: It is extremely simple.)
7. Are any analogous optimizations to the joinable decoder, the factorable decoder, or the back end optimized encoder applicable to transcoders (aside from their implicit use in the internal encoders and decoders), and if so, how beneficial are they?
8. If any generating function derived in this chapter, whether for encoders, decoders, completion detectors, sequencers, majority gates, or transcoders, were at all untrustworthy, what specific machine-checkable conditions comparable to those given in [Section 11.7](#) for decision waits could keep it honest? (hint: Sparse decision waits are restricted decoders. Decoders are generalized sparse decision waits.)
9. Are encoders as simple as they seem? A local optimization depicted in [Figure 13.16](#) reduces the number of FORK and MERGE elements in a circuit fragment without affecting the observable behavior.
- Is the optimization valid, and if so, how could we be sure?
 - Is it ever applicable to a supposedly already optimized encoder $EC(c)$?
 - If a circuit presents multiple opportunities for this transformation, could the order in which they are taken have any adverse effect on performance?
 - Is there an efficient algorithm for generating optimal encoders?

So in all human affairs one notices,
if one examines them closely, that it
is impossible to remove one
inconvenience without another
emerging.

Niccolo Machiavelli

CHAPTER

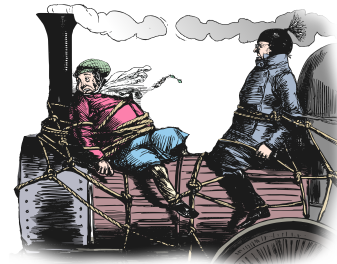
14

WORKING ON THE RAILROAD

As noted in [Chapter 13](#), two of the most useful types of delay insensitive codes are dual rail codes and optimal k -of- n codes (also called Sperner codes hereafter) for different reasons. Dual rail codes enable straightforward arithmetic and logical operations, whereas Sperner codes optimize the channel capacity while in transit. These encodings need not be mutually exclusive. For example, a system consisting of multiple chips connected by a communication network could use the dual rail form on each chip but the Sperner coded form on the network. However, to realize the advantage of such a scheme, methods of converting between the two encodings are needed. These conversions are the subject of this chapter.

Theoretically the problems of transcoding in either direction between dual rail and Sperner codes are already solved as special cases of [Equation 13.22](#), but there are several reasons for taking an interest in custom solutions. The main reason is space efficiency. The space needed by the construction in [Equation 13.22](#) grows at least linearly with the code size, to the point that using it for something like a 32 bit code would be absolutely infeasible, whereas a code of this size could well be amenable to the methods described in this chapter. Another reason is to gain a sense of how delay insensitive circuits involving arithmetic operations and realistic data paths get things done. A third reason is that this area historically has attracted its share of trivial and overly broad patents (citations left as an exercise). While some of these fortunately have expired or are nearing expiration at this writing, cultivating a communal knowledge base about dual rail and Sperner coding is a prudent mitigation against similar litigations in the future.

Unlike most previous chapters, this one does not require any substantial new theory or mathematical notation to be introduced, but does require wielding it effectively. We start small by developing dual rail unsigned integer adders, subtractors, and buffers in [Section 14.1](#), a task fortunately made even smaller by restricting attention to single operands with hard wired constants added or sub-



tracted. The buffers and subtracters are needed for a family of dual rail to Sperner code conversion networks specified in [Section 14.2](#) that work by implementing a simple recursive algorithm. A specification for a family of Sperner to dual rail conversion networks developed in [Section 14.3](#) uses the adders to implement something like an inverse of this algorithm, and in [Section 14.4](#) we investigate a more concurrent implementation.

14.1 Arithmetic units

The three kinds of arithmetic units needed for the transcoders in this chapter have certain features in common. Each of them is made of a cascade of standard cells, with each cell built around a 2-by-2 decision wait and dual rail channels in and out. The adder and the subtracter cells interact with neighboring cells, but the buffer cells share common control and status signals in parallel. A further similarity between the adder and the subtracter is that they are both made from essentially the same standard cell shown in [Figure 14.1](#) and described by the expression



$$AC = Z^2R(\text{FORK}, ZR(Z^4R(\text{DW}(2, 2), (ZR(\text{FORK}, \text{MERGE}))^2 \uparrow 1), \text{MERGE})) \quad (14.1)$$

with its input and output signals ordered and interpreted slightly differently depending to the context. Discussions of how these circuits specifically perform addition, subtraction, and buffering follow respectively in [Section 14.1.1](#), [Section 14.1.2](#), and [Section 14.1.3](#).

14.1.1 Adders

An adder implements the same algorithm for addition taught in elementary school, but with binary numbers instead of decimal. That is, two numbers are added bit by bit starting with their respective least significant bits, each bitwise addition yields a sum bit and a carry out bit, and the carry out is added together with the next two least significant bits. When its value is 1, the carry out represents that the sum has overflowed the current place, for example because 1 plus 1 is 10 in binary arithmetic, which is too wide to fit into a single bit.

The work of executing this algorithm in hardware is allocated to a cascade of cells as mentioned above, with one cell for each pair of bits to be added. Consequently, each cell has a sum and a carry out as outputs, data as inputs, and maybe also a carry input if the cell is not the lowest. Before designing the circuit, we can get an overview of what its inputs and outputs should be by tabulating them as in [Table 14.1](#). By way of a quick check, the carry out and sum entries in each row should express the count of 1's in the rest of the row as a binary number ranging from 0 to 11.

Any textbook on logic design includes a discussion of adders comparable to the foregoing [[38](#), [135](#), [175](#)], but we have to take it in a different direction from here. One difference is that the data, carries, and sums for our adders are transmitted in dual rail form. In other words, a data input is conveyed via two lines, labeled \bar{x} and x in [Figure 14.1](#), with the understanding that a signal received on \bar{x} is interpreted as a data value of 0, and a signal on x as a 1. The same convention applies to the other input and output labels.

Another difference is that our adders take only one operand, with the other operand h from [Table 14.1](#) being a hard wired constant. Consequently, we need two kinds of adder cells, one that adds 0 to the data input when the corresponding bit of h is 0, and the other that adds 1 to it when

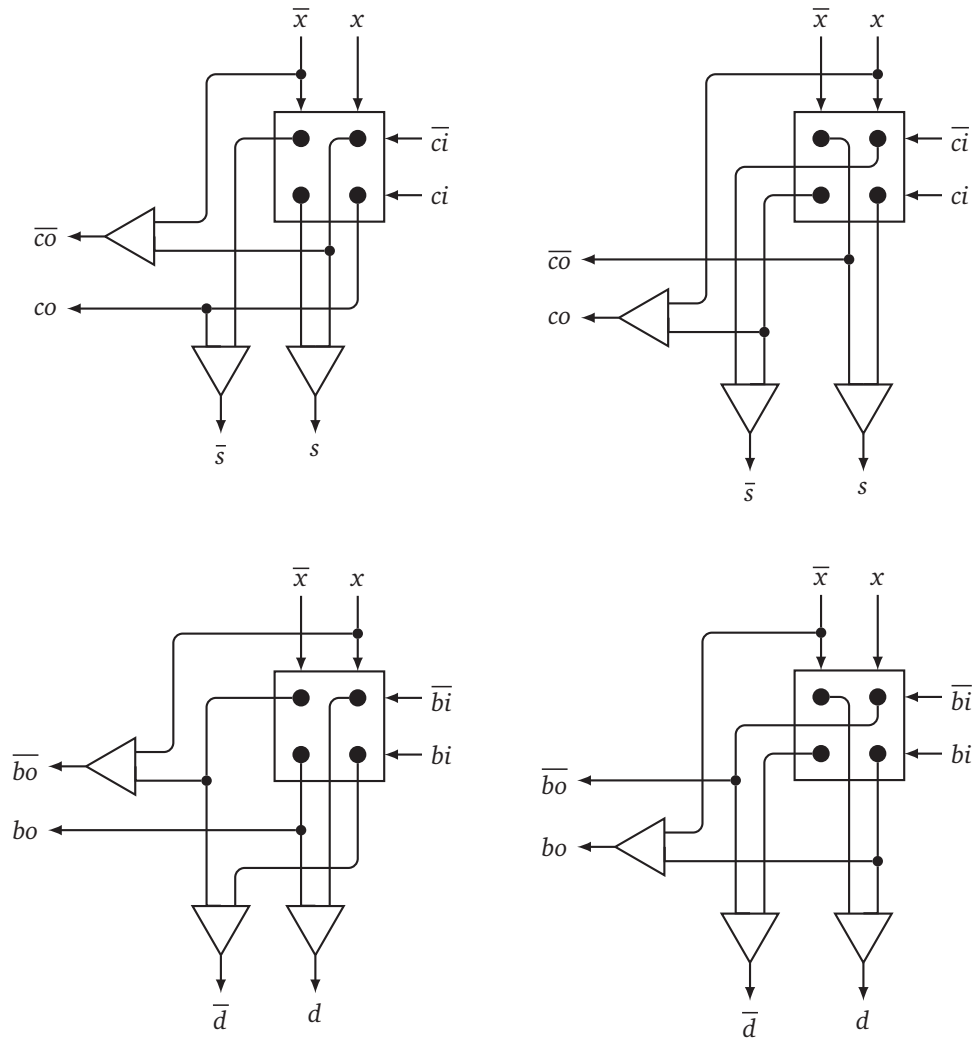


Figure 14.1: An adder of zero (upper left) and an adder of one (upper right) have carries in and carries out, while a subtractor of zero (lower left) and a subtractor of one (lower right) have borrows in and borrows out. All have one bit of dual rail data in and a sum or difference out.

x	h	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 14.1: specification of a one-bit adder in terms of the sum s and carry out co determined by each combination of addends x and h and carry in ci

the corresponding bit of h is 1. This condition accounts for the distinction between an adder of 0 in Figure 14.1 and an adder of 1.

Given these interpretations, the reader should be able to confirm by manual simulation that both of the adder cells in Figure 14.1 behave consistently with Table 14.1. Depending on their experience and outlook, it might also either delight or chagrin readers familiar with synchronous logic design to note that an adder made from a cascade of these cells would be of the *carry lookahead* variety, because it does not wait for the carry in before generating the carry out when adding 0 to 0 or 1 to 1. Carry lookahead adders are considered more performant than the alternative of *ripple carry* adders, but designing a carry lookahead adder in conventional synchronous logic is extremely painful by almost any means other than retrieving it from a component library designed by someone else.

Edge cases

To obtain any member of a family of adders, we seek a function $\text{ADD} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a pair of natural numbers $(k, n) \in \mathbb{N} \times \mathbb{N}$ to a circuit $\text{ADD}(k, n) \in \mathbb{H}$ with a dual rail input bus of the minimum width sufficient to communicate a natural number up to n inclusive and an output bus of the minimum width sufficient to emit $n + k$ in dual rail form. Unfortunately these simple requirements already indicate a few worrisome edge cases neglected by the discussion above.

- Because the output bus is always wide enough for outputs up to $n + k$, the adder should never overflow, so there is no need for a carry out from the most significant bit cell. However, if the output bus is exactly one bit wider than the input bus, then the carry out from the cell computing the second most significant bit of the sum can be used as the most significant bit of the sum.
- If n is not the predecessor of a power of 2, then numbers greater than n are possible to transmit to the adder on the input bus, but no particular output need be specified for those cases and divergence would be acceptable.
- If k is even, or more generally if k satisfies $k \bmod 2^w = 0$ for a positive w , then we should save the cost of w adder cells for the low order bits by replacing them with dual rail channels connecting the data input lines directly to the sum output lines.

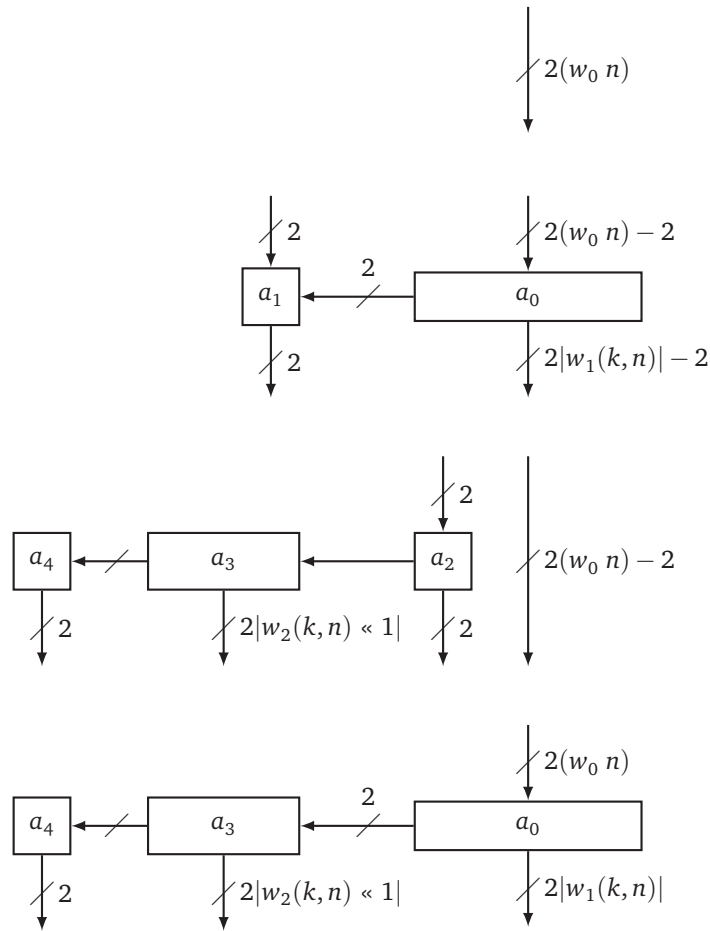


Figure 14.2: Four ways an adder of a constant k to a dual rail coded number up to n might turn out are for $k = 0$ (top), for input and output buses of equal width (second from top), for k wider than n with all low order bits 0 (third), and for k wider than n in general (bottom). Blocks are labeled by functions a_0 through a_4 responsible for creating them.

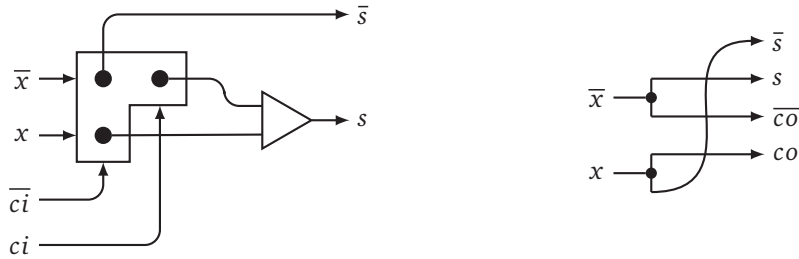


Figure 14.3: An adder of 0 cell $L_2\langle LJOIN, MERGE \rangle$ with no carry out (left) and an adder of 1 cell $FORK^2 \downarrow 1$ with no carry in (right).

- If k exceeds n sufficiently, then some cells in the adder computing the high order bits lack data inputs but must still output a sum dependent on the carries in. A carry out from some lower order cells is needed in this case even if all low order bits of k are 0, perhaps limiting the scope of the optimization proposed above.

Bus widths

A way of allowing for each of these possibilities is to define the ADD function in four cases corresponding to those illustrated by Figure 14.2 in terms of the annotations given by w_0 through w_2 as shown. The function $w_0 : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$w_0 = \lambda n. \lceil \log_2(n + 1) \rceil + \delta_0^n \tag{14.2}$$

takes a number $n \in \mathbb{N}$ to the number of bits $w_0 n \in \mathbb{N}$ in a dual rail channel sufficient to transmit any number up to n . The expression $w_1(k, n) \in \{0, 1\}^{w_0 n}$ represents a suffix of the list of the bits in $k \in \mathbb{N}$ viewed as binary number, with the least significant bit last in the list, and the length of the list either restricted to the width $w_0 n$ or padded out to it with leading zeros depending on which of k or n is wider.

$$w_1 = \lambda(k, n). (\lambda j. (\lambda i. \lfloor k2^{i+1-j} \rfloor \bmod 2)^* t_j) w_0 n \tag{14.3}$$

The expression $w_2(k, n) \in \{0, 1\}^{w_0(n+k) - w_0 n}$ represents a list of any remaining bits of k in order of decreasing significance, of which some are necessarily non-zero if k exceeds n , and any leading zeros needed to pad it out to a width of $w_0(n + k) - w_0 n$ regardless.

$$w_2 = \lambda(k, n). w_1(k, n + k) \upharpoonright w_0(n + k) - w_0(n) \tag{14.4}$$

Equal bus widths

The simplest case for the adder is with $k = 0$, for which $ADD(0, n)$ is just a bus $I^{2(w_0 n)}$, so we can pass immediately to the next simplest, which is with input and output buses of exactly the same width $2w_0(n) = 2w_0(n + k)$ due to a small positive value of k , and hence $|w_2(k, n)| = 0$ by Equation 14.4. This condition implies that $w_1(k, n)_0$, the most significant bit of k padded out to the width of n , must be zero, or else the output bus would have to be one bit wider than the input to hold $n + k$, so the cell computing the most significant bit of the sum is an adder of 0 with no carry out as shown at the left of Figure 14.3. The behavior of this cell is unspecified when the data and carry in are both

1, but this possibility is precluded for all operands up to n so the effect is a matter of indifference as discussed above. In addition to this cell, the two adder cells shown in Figure 14.1, and a dual rail channel I^2 for low order 0 bits as discussed previously, the cell $\text{FORK}^2 \downarrow 1$ implementing an adder of 1 with no carry in shown at the right of Figure 14.3 is needed for the lowest order non-zero bit in k .

Ordering the terminals of the adder of 0 cell from Figure 14.1 consistently with those of the cells in Figure 14.3 requires an expression like

$$\langle 0, 3, 1, 2 \rangle \times_{\text{AC}} \times \langle 2, 0, 3, 1 \rangle$$

in terms of the general arithmetic cell AC from Equation 14.1, and an expression like

$$\langle 3, 0, 2, 1 \rangle \times_{\text{AC}} \times \langle 0, 2, 1, 3 \rangle$$

for the adder of 1. That is, the inputs are ordered $\bar{x}, x, \bar{c}i, ci$ and outputs are ordered $\bar{s}, s, \bar{c}o, co$ by the permutations shown.

This arrangement of input and output terminals implies that in a cascade of cells forming an adder, the last two outputs of each cell that generates a carry out should be connected to the last two inputs of the one next to it on the most significant bit side. Those that do not generate carries out require no connection to their neighbors on the most significant bit side (if any) and need only be put in parallel with them. An expression $a_0(w_1(k, n) \ll 1)$ accounting for all cells in the cascade except the one computing the most significant bit of the sum therefore follows from a definition of the function $a_0 : \{0, 1\}^* \rightarrow \{0, 2\} \times \mathbb{H}$ as

$$a_0 = F_{(0,Z)} \lambda(h, (c, t)) \cdot \begin{cases} (0, \mathbf{R}(t, I^2)) & \text{if } h = 0 \wedge c = 0 \\ (2, \mathbf{R}(t, \text{FORK}^2 \downarrow 1)) & \text{if } h = 1 \wedge c = 0 \\ (2, \mathbf{L}_2 \langle t, \langle 0, 3, 1, 2 \rangle \times_{\text{AC}} \times \langle 2, 0, 3, 1 \rangle \rangle) & \text{if } h = 0 \wedge c = 2 \\ (2, \mathbf{L}_2 \langle t, \langle 3, 0, 2, 1 \rangle \times_{\text{AC}} \times \langle 0, 2, 1, 3 \rangle \rangle) & \text{if } h = 1 \wedge c = 2 \end{cases} \quad (14.5)$$

which folds a function taking an operand of the form $(h, (c, t))$ over the list $w_1(k, n) \ll 1$, which is the binary representation of k by Equation 14.3. The intuition is that $h \in \{0, 1\}$ in this expression represents the bit of k determining the kind of adder cell appropriate for the current position in the cascade along with the number $c \in \{0, 2\}$ of carry out lines from the cascade due to the lower order cells, and $t \in \mathbb{H}$ is the actual cascade of lower order cells. The resulting circuit has a dual rail input data bus with the least significant bit first, dual rail output bus carrying the sum also with the least significant bit first, and two carry out lines (based on the current condition of not all instances of h being 0).

The complete adder including the most significant bit is expressible as $a_1 a_0(w_1(k, n) \ll 1) \in \mathbb{H}$ in terms of a function $a_1 : \mathbb{N} \times \mathbb{H} \rightarrow \mathbb{H}$ defined by

$$a_1 = \lambda(c, t). \mathbf{L}_2 \langle t, \text{LJOIN, MERGE} \rangle$$

which connects the cell on the left of Figure 14.3 to the result above.

Unequal bus widths with low order bits all zero

In all other cases, the output bus width $2w_0(n+k)$ of an adder exceeds the input bus width $2(w_0 n)$. The n least significant bits of the sum can still be computed as above, but the rest of the sum has to be inferred somehow by a cascade of cells with no input data.

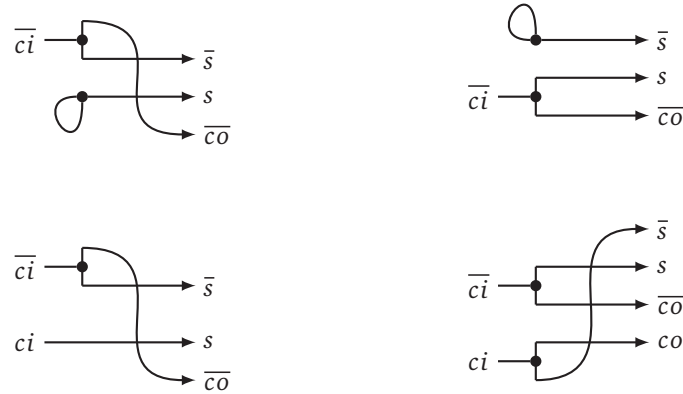


Figure 14.4: Among adders of 0 (left) and adders of 1 (right) with no data in and either half carries in (above) or full carries in (below), only the adder of 1 with a full carry in needs a full carry out.

An especially odd situation is that of all low order bits of k up to the width of n being zero, as shown second from the bottom of Figure 14.2. Passing all w_0 n low order bits through a bus in this case would yield a correct result for the low order bits of the sum, but would give no signal to the rest of the adder to generate the high order bits. The figure shows a solution to this problem by passing all but the top one of the low order bits straight through a bus and taking a carry out from the top one on its way through to control the rest of the adder. Because the carry out from adding a word of zeros to anything is necessarily zero, only one carry out line is needed instead of the usual dual rail form, which can come from a MERGE of the data lines. An expression describing the whole block including the bus is given by $a_2 w_0 n$ for a function $a_2 : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{H}$ defined as

$$a_2 = \lambda w. (1, \mathbf{R}(1^{2(w-1)}, \mathbf{L}_2 \langle \mathbf{FORK}^2 \uparrow_2^1, \mathbf{MERGE} \rangle))$$

where the left side 1 in the result indicates that there is only one carry out line instead of the usual two, hereafter called a **half carry** for the sake of discussion, as opposed to a **full carry** transmitted on two lines.

The high order bits of a sum with no data inputs and only a single carry in that is always zero are necessarily constant, but it is just as well to approach the rest of the design like that of a modified adder built from a cascade of cells based on the simplifying assumption of input data being always zero. That is, we have one type of cell that adds 0 to its inferred input of 0, and the other that adds 1, in either case with a half carry in and a half carry out as shown on the top row of Figure 14.4. These cells are expressible respectively as $\mathbf{R}(\mathbf{FORK}, \mathbf{Z FORK}) \uparrow 1$ and $\mathbf{R}(\mathbf{Z FORK}, \mathbf{FORK})$.

Formally the block $\mathbf{Z FORK}$ has an input arity of zero and a single output terminal that never emits a signal. This specification evokes the idea of a floating terminal but it could also be implemented as a grounded terminal depending on whatever is appropriate for the technology. An even better idea is to regard it as a hook for subsequent local optimizations whereby its combination with a MERGE is replaced by a wire (cf. item 9, page 448).

These cells suffice for computing the rest of the sum except for the most significant bit, which must be handled in some other way to avoid generating a carry out. Restricting attention therefore to

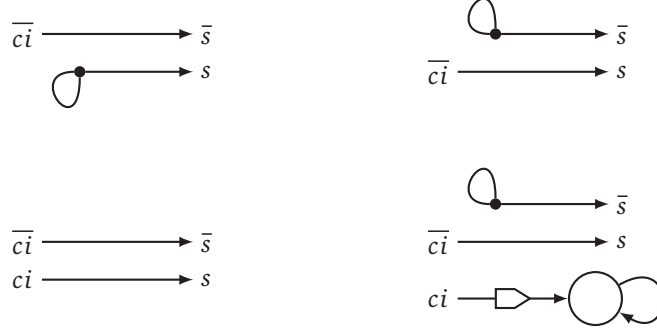


Figure 14.5: Four types of adder cells with no data in and no carries out are adders of 0 (left column) and adders of 1 (right column), with either half carries in (top row), or full carries in (bottom row).

$w_2(k, n) \ll 1$, which includes all high order bits in k except the most significant by Equation 14.4, we can evaluate the cascade $(a_3 a_2 w_0 n) (w_2(k, n) \ll 1)$ by a function $a_3 : \mathbb{N} \times \mathbb{H} \rightarrow (\{0, 1\}^* \rightarrow (\mathbb{N} \times \mathbb{H}))$ defined as

$$a_3 = \lambda b. f_b \lambda(h, (c, t)). \begin{cases} (1, L_1 \langle t, R(\text{FORK}, Z \text{ FORK}) \uparrow 1 \rangle) & \text{if } h = 0 \wedge c = 1 \\ (1, L_1 \langle t, R(Z \text{ FORK}, \text{FORK}) \rangle) & \text{if } h = 1 \wedge c = 1 \\ (1, L_2 \langle t, R(\text{FORK}, I) \uparrow 1 \rangle) & \text{if } h = 0 \wedge c = 2 \\ (2, L_2 \langle t, \text{FORK}^2 \downarrow 1 \rangle) & \text{if } h = 1 \wedge c = 2 \end{cases} \quad (14.6)$$

which folds a function that operates on a tuple $(h, (c, t))$ with the same interpretation as in Equation 14.5 over the list of bits $w_2(k, n) \ll 1 \in \{0, 1\}^*$ using $b = a_2 w_0 n \in \mathbb{N} \times \mathbb{H}$ as the base. This function also copes with full carries using the cells in the lower row of Figure 14.4. Although they are not needed in the current case of $w_1(k, n) \in \{0\}^*$, they become relevant shortly.

One more cell would complete the construction of the adder by computing the most significant bit of the sum. Depending on the most significant bit of the constant k as given by $w_2(k, n)_0$, it should transmit either its carry in or the complement thereof to the sum as shown in Figure 14.5, and generate no carry out of its own. The result $a_4(w_2(k, n)_0, (a_3 a_2 w_0 n) (w_2(k, n) \ll 1))$ expresses the connection of the appropriate cell in Figure 14.5 to the cascade for the current case of $w_1(k, n) \in \{0\}^*$ when the function $a_4 : \{0, 1\} \times (\{1, 2\} \times \mathbb{H}) \rightarrow \mathbb{H}$ is defined as follows

$$a_4 = \lambda(h, (c, t)). \begin{cases} L_1 \langle t, R(I, Z \text{ FORK}) \rangle & \text{if } h = 0 \wedge c = 1 \\ L_1 \langle t, R(Z \text{ FORK}, I) \rangle & \text{if } h = 1 \wedge c = 1 \\ L_2 \langle t, I^2 \rangle & \text{if } h = 0 \wedge c = 2 \\ L_2 \langle t, R(R(Z \text{ FORK}, I), Z^2 R(\text{PUSH}, \text{JOIN})) \rangle & \text{if } h = 1 \wedge c = 2 \end{cases} \quad (14.7)$$

with h, c , and t interpreted as above. To make it more general, this function is also defined for full carries in. However, an adder of 1 with a full carry in and no carry out overflows when the carry in is 1. This event should never be possible with correctly chosen bus widths, so as an aid to formal verification (also known as a sanity check) the cell is deliberately constructed to diverge under those circumstances by laying a trap for the carry in (cf. Figure 11.1).

Unequal bus widths with low order bits not all zero

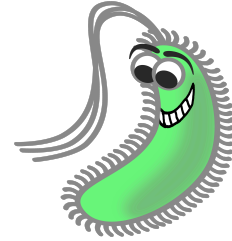
The last case for an adder specification depicted at the bottom of [Figure 14.2](#) pertains to an output bus width $2w_0(n+k)$ exceeding the input bus width $2(w_0 n)$ along with the assumption of at least one non-zero bit in the range of $w_1(k, n)$, the low order bits of k up to the width of n . In this case we can construct the first stage of the adder as $a_0 w_1(k, n)$ by [Equation 14.5](#) instead of $a_0(w_1(k, n) \ll 1)$ as required for equal bus widths, and use it in the expression $(a_3 a_0 w_1(k, n)) (w_2(k, n) \ll 1)$ in place of $a_2 w_0 n$, which is needed only for low order bits all zero, to express the rest of the adder except for the most significant bit cell. The full carry out associated with the cascade given by $a_0 w_1(k, n)$ necessitates the latter two cases in the definition of a_3 by [Equation 14.6](#). The rest of the construction is completed as above by $a_4(w_2(k, n)_0, (a_3 a_0 w_1(k, n)) (w_2(k, n) \ll 1))$ in terms of the function a_4 defined by [Equation 14.7](#). The cases in this definition pertaining to full carries are relevant here as well.

In summary, we have the following definition for the function $\text{ADD} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ to cover all cases. Where these conditions overlap, the first that applies is preferable.

$$\text{ADD}(k, n) = \begin{cases} 1^{2(w_0 n)} & \text{if } k = 0 \\ a_1 a_0(w_1(k, n) \ll 1) & \text{if } |w_2(k, n)| = 0 \\ a_4(w_2(k, n)_0, (a_3 a_2 w_0 n) (w_2(k, n) \ll 1)) & \text{if } w_1(k, n) \in \{0\}^* \\ a_4(w_2(k, n)_0, (a_3 a_0 w_1(k, n)) (w_2(k, n) \ll 1)) & \text{otherwise} \end{cases}$$

14.1.2 Subtracters

Subtracters of constant subtrahends k can be organized similarly to adders as cascades of cells wherein each cell is hard wired to subtract one bit of k from the corresponding bit of the input data. Instead of sending a carry out, each cell sends a borrow signal to its neighboring cell or to the environment. Unlike adders, subtracters always have a borrow output from the most significant bit position. The output data bus is never wider than the input, but it can be narrower or even empty if k is large, in which case at least some of the cells have data and borrows in but no data out. The subtracter function $\text{SUB} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a pair of numbers $(k, n) \in \mathbb{N} \times \mathbb{N}$ to a block $\text{SUB}(k, n) \in \mathbb{H}$ needs to cope with at least the four cases illustrated in [Figure 14.6](#).



- If k is zero and n is positive, then the subtracter only needs to pass the data unaltered to the output and generate a borrow out that is always zero.
- If k is equal to n , there is no data bus out because the difference can be at most zero, and the borrow out always transmits a value of one unless the input is exactly n .
- If all low order bits of k up to the width of the maximum possible output $n - k$ are zero, a half borrow derived from one of the low order bits connects to the borrow propagation network generated by s_1 , and the low order bits pass on a bus to the output.
- In the general case, bits of the difference up to the width of $n - k$ are computed by subtracter cells, whose top borrow out reaches the borrow propagation network if applicable.

The useless case of $k > n$ is safe to ignore, but because $k = n$ is allowed, it is only fitting to handle the case of $k = n = 0$ consistently with $k = n$ in general, which is with an empty output data

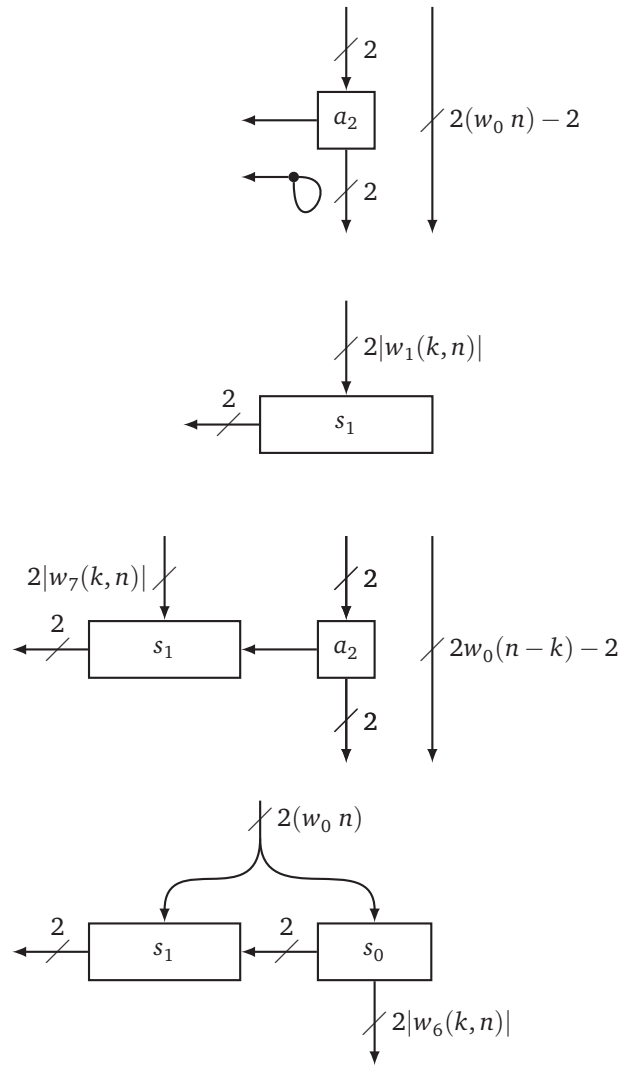


Figure 14.6: Four cases of a subtractor of a constant $k < n$ from a dual rail coded number up to n for $k = 0$ (top), for $k = n$ (second from top) for low order bits of k equal to 0 (third), and for general k (bottom) reusing a_2 from the adder with new functions s_0 and s_1 (cf. Figure 14.2).

x	h	bi	bo	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 14.2: specification of a one-bit subtracter in terms of the difference d and borrow out bo determined by each combination of data x , constants h and borrows in bi (cf. Table 14.1)

bus. A circuit of the form

$$(\neq \mathbf{R}) \langle \mathbf{I}, \mathbf{Z}^2 \mathbf{R}(\text{PUSH}, \text{JOIN}), \mathbf{Z} \text{ FORK} \rangle$$

similar but not identical to one in Figure 14.5 covers this case by having two input lines \bar{x} and x of which the second is prohibited and two output lines \bar{bo} and bo of which the second never sends a signal, so that a signal to \bar{x} representing a zero data input causes a signal on \bar{bo} representing a zero borrow out.

To construct the cells needed for subtraction in general, we can start by consulting Table 14.2, which displays the borrow out and difference bits corresponding to each combination of data, constant bits, and borrows in according to the usual algorithm for unsigned binary subtraction. The table shows that subtraction is not far from addition, with the only change being that the borrow out is the complement of what the carry out would be for an adder. Complementing a dual rail signal is achievable by interchanging the bus lines, so specifications for the subtracter cells in Figure 14.1 are obtained relatively easily by switching the last two numbers of the output permutations of the adder cells used in Equation 14.5. The subtracter of 0 cell is therefore

$$\langle 0, 3, 1, 2 \rangle \times \text{AC} \times \langle 2, 0, 1, 3 \rangle$$

and the subtracter of 1 cell is

$$\langle 3, 0, 2, 1 \rangle \times \text{AC} \times \langle 0, 2, 3, 1 \rangle.$$

These cells have a full borrow in, but a cascade implementing a subtracter would need at least one cell with no borrow in for the least significant bit. If the least significant bit of the constant is 0, then that cell needs no borrow out and can be implemented by a pair of wires \mathbb{I}^2 as in an adder. No borrow in or out is involved therefore from the least significant bit position up to the position of the least significant 1 bit in the constant. A subtracter of 1 with no borrow in is given by $\text{FORK}^2 \times \langle 2, 0, 3, 1 \rangle$ based on the corresponding case for an adder in Equation 14.5 with the last two output lines interchanged (cf. Figure 14.3), and has a full borrow out. Based on these four cases, a function $s_0 : \{0, 1\}^* \rightarrow \{0, 2\} \times \mathbb{H}$ analogous to a_0 generating a subtracter as a cascade of cells is

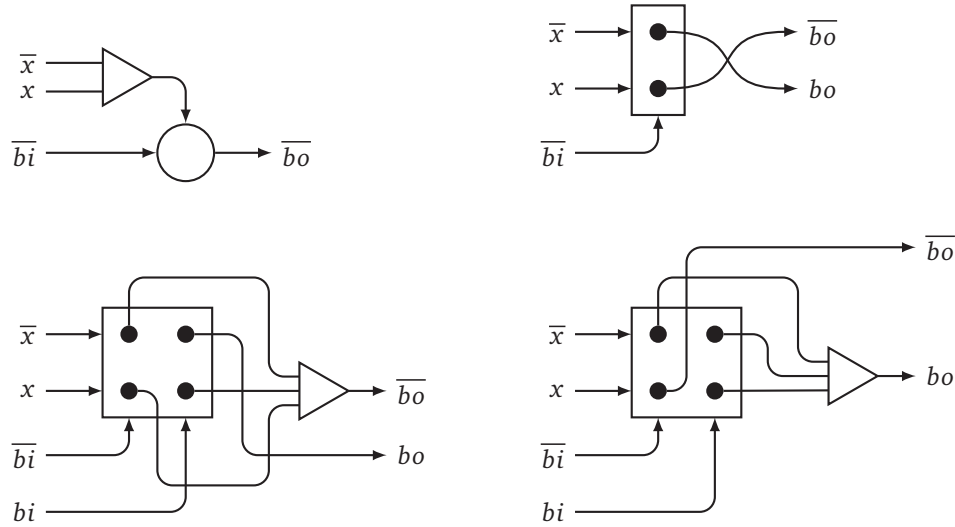


Figure 14.7: A cascade of subtracters of 0 (left column) and 1 (right column) with half borrows in (top row) or full borrows in (bottom row) but no data out forms the borrow propagation network.

expressible as follows.

$$s_0 = F_{(0,zi)} \lambda(h, (b, t)) \cdot \begin{cases} (0, \mathbf{R}(t, I^2)) & \text{if } h = 0 \wedge b = 0 \\ (2, \mathbf{R}(t, \text{FORK}^2 \times \langle 2, 0, 3, 1 \rangle)) & \text{if } h = 1 \wedge b = 0 \\ (2, \mathbf{L}_2 \langle t, \langle 0, 3, 1, 2 \rangle \times \text{AC} \times \langle 2, 0, 1, 3 \rangle \rangle) & \text{if } h = 0 \wedge b = 2 \\ (2, \mathbf{L}_2 \langle t, \langle 3, 0, 2, 1 \rangle \times \text{AC} \times \langle 0, 2, 3, 1 \rangle \rangle) & \text{if } h = 1 \wedge b = 2 \end{cases}$$

The output data bus of this cascade should be only as wide as necessary to carry the maximum possible difference $n - k$ in dual rail form, which may be less than the width of input bus needed to carry the input n . To restrict it to this width, we should apply s_0 only to the low order bits

$$w_6(k, n) = w_1(k, n) \ll (w_0 n) - w_0(n - k)$$

of the list $w_1(k, n)$ by Equation 14.3 (assuming $k < n$) and make other arrangements for the high order bits

$$w_7(k, n) = w_1(k, n) \uparrow (w_0 n) - w_0(n - k).$$

The rest of the subtracter in the general case takes the form of a borrow propagation network having an input data bus fed by the high order bits $w_7(k, n)$, no output data bus, and a full borrow out. This network can also be specified as a cascade of cells similar to those considered already, but simpler due to the lack of data outputs, in terms of a function s_1 yet to be defined. As shown in Figure 14.6, the borrow propagation network appears in three contexts where it might have either no borrow in, a half borrow in, or a full borrow in, which along with two possible subtrahend bits makes six possible cells to consider in its definition.

- An easy case is a subtracter of 0 with no borrow in. Neither possible input incurs any debt, so the cell reduces to a MERGE whose inputs are \bar{x} and x and whose output maps to \bar{b}_o , the half borrow out.

- Also pretty easy is a subtracter of 1 with no borrow in, which has a full borrow out given by the complement of the data in. This cell only needs to interchange two wires in a circuit of the form $I^2 \downarrow 1$.
- The two cases with half borrows in are similar to the foregoing except insofar as the data are synchronized with the borrow in before the cell propagates the half or full borrow out to its neighbor as shown along the top row of [Figure 14.7](#).
- The last two cases require a decision wait and a MERGE to generate the same borrow for three out of four combinations. A subtracter of 0 never borrows unless the data bit is 0 and the borrow in is 1, while a subtracter of 1 always borrows unless the data bit is 1 and the borrow in is 0. These cases are shown on the bottom row of [Figure 14.7](#).

In light of these observations, a construction of a function $s_1 : \{0, 1, 2\} \times \mathbb{H} \rightarrow (\{0, 1\}^* \rightarrow \mathbb{H})$ by six cases follows directly.

$$s_1 = \lambda c. (\lambda(b, t). t) \circ \mathcal{F}_c \lambda(h, (b, t)). \begin{cases} (1, \text{MERGE}) & \text{if } h = 0 \wedge b = 0 \\ (2, I^2 \downarrow 1) & \text{if } h = 1 \wedge b = 0 \\ (1, L_1 \langle t, ZR(\text{MERGE}, \text{JOIN}) \rangle) & \text{if } h = 0 \wedge b = 1 \\ (2, L_1 \langle t, DW(2, 1) \downarrow 1 \rangle) & \text{if } h = 1 \wedge b = 1 \\ (2, L_2 \langle t, L_3 \langle DW(2, 2) \uparrow 1, \text{MERGE } 3 \rangle \downarrow 1 \rangle) & \text{if } h = 0 \wedge b = 2 \\ (2, L_2 \langle t, F_3 \langle DW(2, 2) \downarrow 1, \text{MERGE } 3 \rangle \rangle) & \text{if } h = 1 \wedge b = 2 \end{cases}$$

It is convenient to make s_1 a second order function parameterized by a pair $c \in \{0, 1, 2\} \times \mathbb{H}$ specifying the subtracter of the lower order bits along with the width of the borrow in. As shown in [Figure 14.6](#), this parameter could be $a_2 w_0(n - k)$ or $s_0 w_6(k, n)$ if there are any lower order bits, or $(0, ZI)$ if not, and would serve as the base for folding a function over the list of high order bits $w_7(k, n)$ or the whole list $w_1(k, n)$ respectively. The composition with $\lambda(b, t). t$ extracts the network and discards the borrow width from the result, so writing $(s_1 a_2 w_0 n) \epsilon$ would be a way of extracting it from the result of $s_1 a_2 w_0 n$ suitably for the case of $k = 0$ and $n \neq 0$. Consequently, we have the following specification of a function to generate subtracters.

$$\text{SUB}(k, n) = \begin{cases} (\mathcal{F} R) \langle I, Z^2 R(\text{PUSH}, \text{JOIN}), Z \text{ FORK} \rangle & \text{if } k = 0 \wedge n = 0 \\ R((s_1 a_2 w_0 n) \epsilon, Z \text{ FORK}) & \text{if } k = 0 \wedge n \neq 0 \\ s_1(0, ZI) w_1(k, n) & \text{if } k \neq 0 \wedge k = n \\ (s_1 a_2 w_0(n - k)) w_7(k, n) & \text{if } k \neq 0 \wedge k \neq n \wedge w_6(k, n) \in \{0\}^* \\ (s_1 s_0 w_6(k, n)) w_7(k, n) & \text{otherwise} \end{cases} \quad (14.8)$$

14.1.3 Buffers

The last prerequisite for a dual rail to Sperner transcoder is a dual rail buffer made from an array of cells such as the one shown in [Figure 14.8](#). The buffer supports two types of interaction with its environment. One type has the environment write dual rail data to the buffer and receive an acknowledgment from it, and the other has the environment send a read request to the buffer and receive dual rail data from it in return. The buffer implements a limited form of storage by returning the same data most recently written to it in response to the first subsequent read request. However, because it is only a buffer and not a register, reading from it also erases its contents, so any subsequent read requests always cause it to return zero until after the next write. Multiple writes without an intervening read are also allowed and duly acknowledged, which cause previously written data to be overwritten by new data.

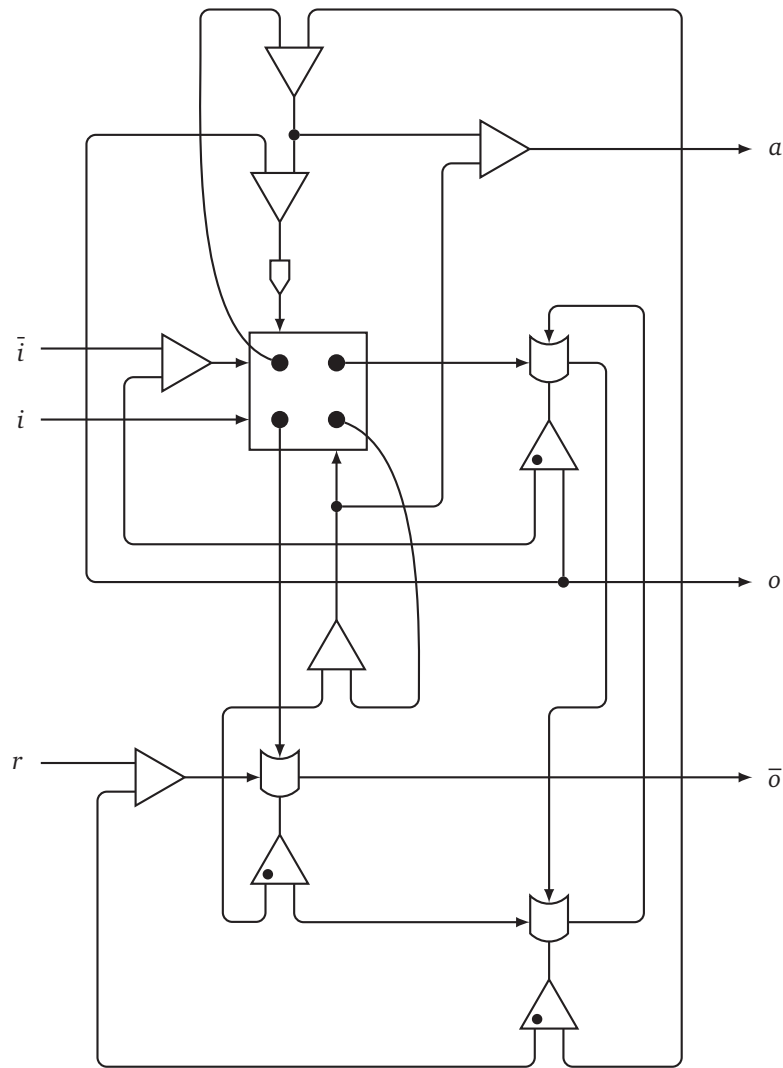


Figure 14.8: A dual rail buffer cell, BU, initially holds a zero bit and is cleared by every read. It has a pair of dual rail write inputs \bar{i} and i , a read request input r , a pair of dual rail data outputs \bar{o} and o , and an acknowledgment output a (in that order).

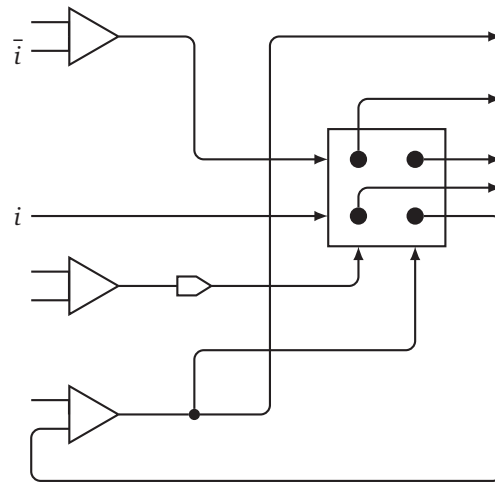


Figure 14.9: $BU_0 = Z(F_4(\bar{f} R \langle \text{MERGE}, I, ZR(\text{MERGE}, \text{PUSH}), ZR(\text{MERGE}, \text{FORK}) \rangle, DW(2, 2)) \downarrow 1)$

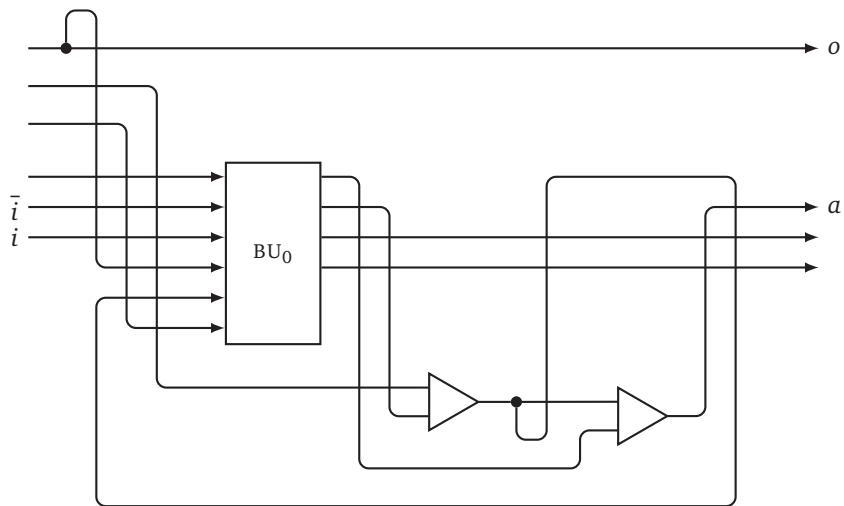


Figure 14.10: $BU_1 = ZR(\text{FORK}, Z((Z^2 R(BU_0, F \langle \text{MERGE}, \text{FORK}, \text{MERGE} \rangle)) \downarrow 2))$

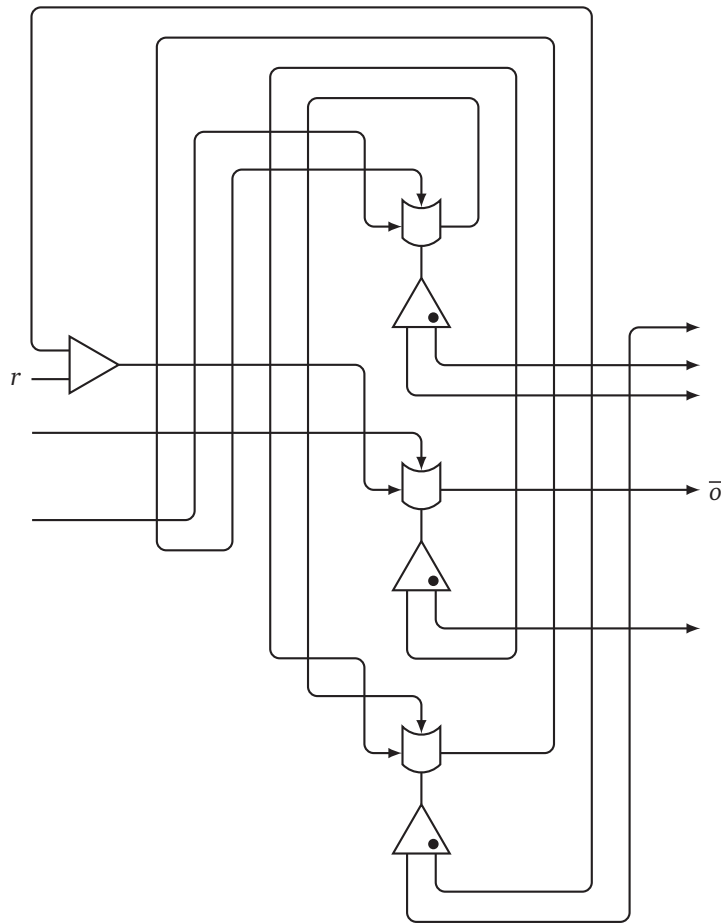


Figure 14.11: $BU_2 = Z(F\langle MERGE, Z((Z((Z(L\langle SHUNT, TOGGLE\rangle^3)) \uparrow 4)) \uparrow 2)) \uparrow 1)$

Theory of operation

Figure 14.8 shows a schematic for the dual rail buffer cell. The main idea is to use a 2-by-2 decision wait as a state holding device such that a write input of either i or \bar{i} causes an output from the left column when the cell holds a value of zero, and the right column otherwise. Writing the current value maintains this state, and writing the alternative changes it. Holding a value of one also implies that the control input to the SHUNT at the lower left has been signaled to redirect the read input signal from the default \bar{o} output to o , and the rest of the circuit follows from the need to maintain all of these conditions. For an explanation in greater detail, see Appendix F.

Specification

Specifying the buffer cell by block combinators might not be the most fun in the world but is achievable in about four steps. First we describe the decision wait and everything that feeds directly

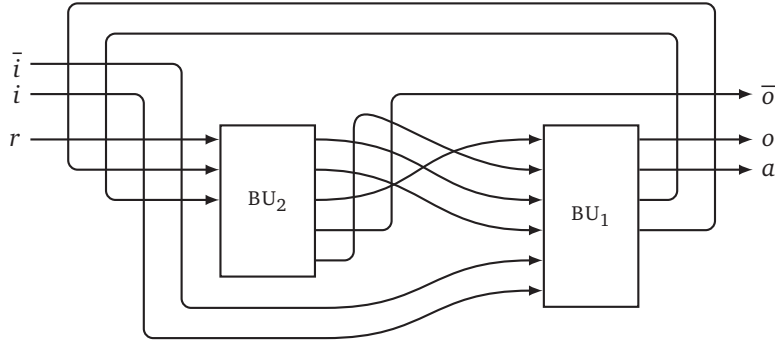


Figure 14.12: $BU = Z^2((BU_2 \downarrow 1 \xrightarrow{\langle 1,2,3,0 \rangle} BU_1) \Downarrow 2)$

into it by the expression

$$BU_0 = Z(F_4(\langle F R \rangle \langle \text{MERGE}, I, ZR(\text{MERGE}, \text{PUSH}), ZR(\text{MERGE}, \text{FORK}) \rangle, DW(2,2)) \downarrow 1) \quad (14.9)$$

as depicted in Figure 14.9. Then we throw on the FORK and the two more MERGE primitives appearing near the top of Figure 14.8 by writing

$$BU_1 = ZR(\text{FORK}, Z((Z^2R(BU_0, F\langle \text{MERGE}, \text{FORK}, \text{MERGE} \rangle)) \Downarrow 2))$$

in terms of Equation 14.9 as depicted in Figure 14.10, which is drafted in keeping with the way a straightforward transcription of this expression might appear. The three SHUNT and TOGGLE combinations, the remaining MERGE, and whatever connections are local to them can be packaged in the expression

$$BU_2 = Z(F\langle \text{MERGE}, Z((Z((Z(L\langle \text{SHUNT}, \text{TOGGLE} \rangle^3)) \uparrow 4)) \uparrow 2)) \uparrow 1)$$

as depicted in Figure 14.11, also drafted to reflect the way it is expressed, and then at last we can put them together in the expression

$$BU = Z^2((BU_2 \downarrow 1 \xrightarrow{\langle 1,2,3,0 \rangle} BU_1) \Downarrow 2)$$

as depicted in Figure 14.12 (with the terminals labeled in the correct order) to arrive at the whole buffer cell. Although there is no reason to take it on faith, a circuit specification this size fortunately is an ideal candidate for automated verification against a process level description as explained at length in Part II of this book.

One last consideration about buffers is that of putting individual buffer cells together into a dual rail buffer capable of storing words of arbitrary width. An n -bit dual rail buffer would need a single read request broadcast to all cells by a FORK network, and a single write acknowledgment synchronized from all of them by a multi-way JOIN. An array of n buffer cells transformed to

$$BU^n \leftarrow \downarrow_3^2 \downarrow_3^2$$

by Equation 8.55 and Equation 8.56 would help by aggregating the read request lines as the first n inputs and the write acknowledgment lines as the first n outputs. We could then define a function

DBU : $\mathbb{N} \rightarrow \mathbb{H}$ taking a number of bits $n \in \mathbb{N}$ to an n -bit dual rail buffer $\text{DBU}(n) \in \mathbb{H}$ as

$$\text{DBU}(n) = \mathbb{F}_n \langle \text{FORK } n, \text{BU}^n \leftarrow_3^2 \text{L}_3^2, \text{JOIN } n \rangle \quad (14.10)$$

so that the first input to $\text{DBU}(n)$ is the read request, and the first output from it is the collective acknowledgment.

14.2 Dual rail to Sperner code conversion

Having established a precedent in [Section 14.1](#) for circuits that implement simple mathematical algorithms, we might try something more challenging now, such as a circuit that computes a unique word in a constant weight code for any number up to the code size. A dual rail to Sperner transcoder corresponds to the special case for which the weight k of the code is equal to $\lfloor n/2 \rfloor$, where n is the number of lines in the output bus. The input bus would carry a number i ranging from 0 to $c - 1$ inclusive in dual rail form for a code size c not exceeding the binomial coefficient of n and k .

What happens when we start to think seriously about a circuit performing this operation? Try as we might, there is no avoidance of some unused channel capacity on one side or the other. Dual rail channels suit code sizes that are powers of two, k -of- n channels suit code sizes that are binomial coefficients, and never the twain shall meet. For example, an 8 bit dual rail channel can transmit numbers from 0 to 255 for a code size of 256, which is slightly more than a 5-of-10 channel with a code size of 252, so it would need to be transcoded to at least a 5-of-11 channel on the output, whose code size of 462 is considerably more than necessary, but not quite enough for a 9 bit dual rail channel with a code size of 512, and so on.

A transcoding algorithm therefore is always stuck with at least two cases. Numbers on the low end, for example from 0 to 251 on an eight bit channel, can be transcoded to words selected from only the first 10 of the 11 available output symbols in a 5-of-11 code, whereas the words generated for the remaining numbers 252 to 255 must include the 11th, but only 4 of the other 10 to make up the constant weight of 5. The latter 4 signals in these left over words could be generated by a circuit designed for a 4-of-10 channel and only a 2 bit input bus, which might be small enough to implement naively by [Equation 13.22](#), or maybe even a smaller channel if the left over code words are insufficient to use all 10 lines (*cf.* [Equation 13.2](#)).

The need to allow for two cases persists even if the transcoder makes full use of the output channel and only partial use of the input channel. For a 5-of-10 output channel and an 8 bit dual rail input channel with inputs above 251 prohibited, half of the numbers (*e.g.*, those up to 125 inclusive) could be encoded using only the first 9 of the available 10 lines, but then the other half of them would need the hitherto unused line and only 4 of the rest.

Undoubtedly there are the ingredients of a dual rail to Sperner transcoder in these ideas somewhere. Turning them into an algorithm is the subject of [Section 14.2.1](#), and designing a circuit to implement it occupies [Section 14.2.2](#).



14.2.1 Transcoding algorithm

To generalize from these observations, the conversion of a number a to a word in a k -of- n code involves a test for whether a could be encoded by a k -of- $(n - 1)$ code, which is true whenever it is

less than a code size of

$$s = \binom{n-1}{k}$$

in which case we do that instead. Otherwise, we convert $a - s$ to a word in a $(k-1)$ -of- $(n-1)$ code, and remember to put $n-1$ into the resulting word afterwards. This method suggests a recursive algorithm whose base case is with $k = n$. The k -of- k code contains only the word $\mathcal{R}(t_k)$, which is the set of natural numbers less than k , and which must be the result if a is valid. This algorithm is expressible as a recurrence

$$f_k^n : \left\{ a \in \mathbb{N} \mid a < \binom{n}{k} \right\} \mapsto \mathcal{P}(\mathcal{R}(t_n))$$

defined as follows.

$$f_k^n(a) = \left(\lambda f. f \binom{n-1}{k} \right) \lambda s. \begin{cases} \mathcal{R}(t_k) & \text{if } k = n \\ f_k^{n-1}(a) & \text{if } a < s \\ f_{k-1}^{n-1}(a-s) \cup \{n-1\} & \text{otherwise} \end{cases} \quad (14.11)$$

Output word format

Although this function always maps any input number within its domain to a distinct code word, it has the quirky feature of being non-monotonic with respect to the lexicographic ordering. That is, inputs a and a' satisfying $a < a'$ do not imply that $f_k^n(a)$ lexicographically precedes $f_k^n(a')$. They do not even imply the contrary. For an increasing sequence of inputs, the algorithm actually generates code words in the reverse of the lexicographic order that would be implied by a reversal of the numerical order of the members of the words. For example, a routine calculation of $(f_3^5)^* t_{10}$ yields the list of words

$$\langle \{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}, \{0, 1, 4\}, \{0, 2, 4\}, \{1, 2, 4\}, \{0, 3, 4\}, \{1, 3, 4\}, \{2, 3, 4\} \rangle$$

which would be in reverse lexicographic order if 4 were the smallest number and 0 were the biggest. More formally, if

$$C_c(k, n) = \{w \in \mathcal{P}(\mathcal{R}(t_n)) \mid k = |w|\} \quad (14.12)$$

denotes the constant weight k -of- n code, then a closed form of the recurrence is

$$f_k^n(a) = (\mu \lambda i. n - i - 1) C_c(k, n)^{\circ-1} \binom{n}{k} - a - 1$$

in terms of $C_c(k, n)^{\circ-1} : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$, which maps a lexicographic ordinal to its code word as usual. The definition of the function f_k^n is worth keeping as it is nevertheless because it lends itself to a hardware implementation as shown in [Section 14.2.2](#), but its particular output word format needs to be taken into account for recovering the original data when transcoding in the opposite direction.

Input word format

The input word format is also worth contemplating. Because a circuit that implements f_n^k involves a subtracter, the representation of the numerical input $a \in \mathbb{N}$ must match the representation required

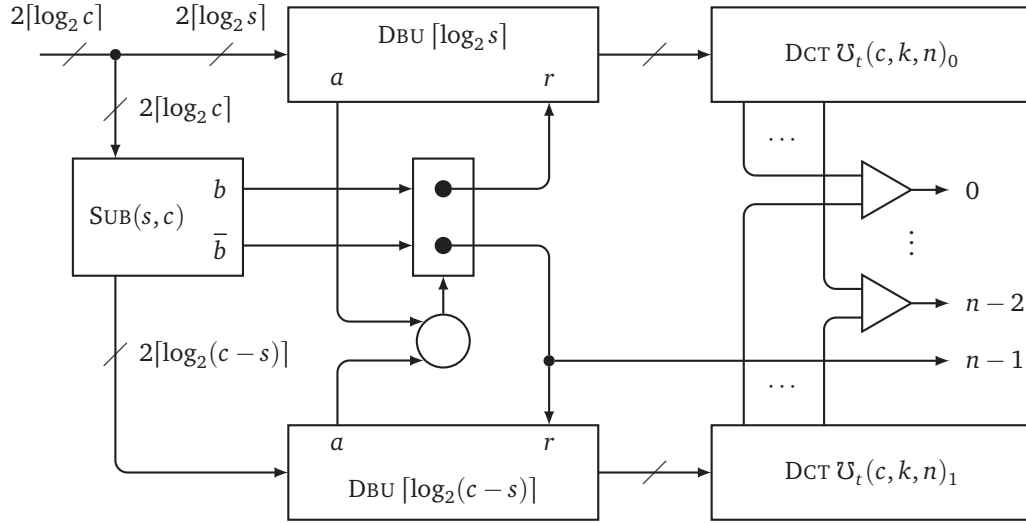


Figure 14.13: A dual rail to k -of- n transcoder $\text{DCT}(c, k, n)$ with code size c , weight k , and bus width n uses a constant subtracter of s , two dual rail buffers, and two smaller transcoders, where s is the binomial coefficient $(n-1)!/(k!(n-1-k)!)$.

by a subtracter according to Section 14.1.2. This representation has the least significant bit first (*i.e.*, starting at bus line number 0) and has the line coding for zero within each pair of lines preceding the other line. As fate would have it, the word representing the number a in this form is not necessarily the a -th word in the code lexicographically, but rather the word

$$\mathcal{R}((\lambda i. 2^i + ([a2^{-i}] \bmod 2))^* \iota_{\lfloor \log_2 c \rfloor}) \in \mathcal{P}(\mathbb{N})$$

for a code size of c and hence $\lfloor \log_2 c \rfloor$ bits in each word.

This expression turns out to be useful in formulating the base case of a recurrence for generating dual rail to Sperner transcoder circuits (*i.e.*, the result for $k=1$, $k=n$, and optionally for other small values of k and n) in terms of the transcoder generating function TC defined by Equation 13.22. A basic transcoder with the same input and output format as the inductive case would be given by

$$\text{TC } \mathcal{R}((\lambda a. (\mathcal{R}((\lambda i. 2^i + ([a2^{-i}] \bmod 2))^* \iota_{\lfloor \log_2 c \rfloor}), f_k^n(a)))^* \iota_c)$$

so as to associate the input word representing the number a with the output word $f_k^n(a)$.

14.2.2 Circuit derivation

A family of circuits implementing a dual rail to constant weight transcoder according to this algorithm and interface specification for a choice of code size c , code weight k , and output bus width n is expressible as $\text{DCT}(c, k, n)$ in terms of a function $\text{DCT} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ defined by a recurrence as suggested above. Along with the aforementioned base case, we have an inductive case

$$\Omega_t(c, k, n, \text{DCT}^* \mathcal{U}_t(c, k, n))$$

following the familiar pattern of a combining form $\Omega_t : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{H}^2 \rightarrow \mathbb{H}$ taking the given parameters and the list of two lower order results to the result overall, and a corresponding decomposition function $\mathcal{U}_t : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow (\mathbb{N} \times \mathbb{N} \times \mathbb{N})^2$ taking the parameters to the list of two tuples of parameters determining each of the lower order results.

$$\text{DCT}(c, k, n) = \begin{cases} \text{TC } \mathcal{R}((\lambda a. (\mathcal{R}((\lambda i. 2^i + \lfloor a2^{-i} \rfloor \bmod 2))^* \iota_{\lfloor \log_2 c \rfloor}, f_k^n(a)))^* \iota_c) & \text{if } (k, n) \in K_s \\ \Omega_t(c, k, n, \text{DCT}^* \mathcal{U}_t(c, k, n)) & \text{otherwise} \end{cases}$$

At a minimum, the set $K_s \subseteq \mathbb{N} \times \mathbb{N}$ of base case parameters used in this recurrence contains all pairs of the form $(1, n)$ and (n, n) for $n \in \mathbb{N}$, and may also contain any others deemed advantageous for implementation reasons. Other necessary conditions for valid results are

- $0 < k \leq n$
- $0 < c \leq \binom{n}{k}$
- $n \leq |\bigcup \mathcal{R}((f_k^n)^* \iota_c)|$

with the last condition being a consequence of [Equation 13.2](#), the constraint on encoders whereby every output line has to be used in at least one word. Fortunately the members of any set of the form $\bigcup \mathcal{R}((f_k^n)^* \iota_c) \subset \mathbb{N}$ are consecutive starting from zero, or else there would be complications.

Decomposition function

A definition for the decomposition function \mathcal{U}_t is attainable based on this condition and a hard look between the lines of [Equation 14.11](#). A circuit that implements f_k^n for a code size c would appear to depend on one that implements f_k^{n-1} for a code size s and one that implements f_{k-1}^{n-1} for a code size $c-s$, except that the output bus width of the latter might need to be adjusted downward from $n-1$ if $c-s$ is small, specifically to $|\bigcup \mathcal{R}((f_{k-1}^{n-1})^* \iota_{c-s})|$ in keeping with the condition above. Hence we have $\mathcal{U}_t(c, k, n) \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N})^2$ formed of the two triples as shown.

$$\mathcal{U}_t(c, k, n) = (\lambda s. \langle (s, k, n-1), (c-s, k-1, |\bigcup \mathcal{R}((f_{k-1}^{n-1})^* \iota_{c-s})|) \rangle) \binom{n-1}{k} \quad (14.13)$$

Combining form

The combining form Ω_t is the interesting part, amounting to a formal description of the circuit shown in [Figure 14.13](#). The way it works is as follows.

- An input word in dual rail form enters at the upper left on a bus of width $2\lceil \log_2 c \rceil$, where c is the code size.
- The $\lceil \log_2 s \rceil$ low order bits of the input word are copied to the buffer at the top by a FORK network while the whole word is fed to the subtracter, and the difference up to $c-s$ goes from the subtracter to the buffer at the bottom.
- If the subtracter asserts a borrow out by the signal labeled b , then the input word must be less than s , so the signal passes through the decision wait to read the low order bits previously stored in the upper buffer after the writes to both buffers are acknowledged via the JOIN.

- If the subtracter transmits the complementary borrow signal labeled \bar{b} , then the input word is not less than s , so the borrow signal causes lower buffer to be read and also propagates itself to the output bus via the FORK. This output signal corresponds to the inclusion of $\{n-1\}$ in the result of Equation 14.11 and indicates that a high number is being transcoded.
- Depending on which buffer is read, either the low order slice of the input word or the representation of the difference between the input word and s is fed to the corresponding transcoder $\text{DCT } \mathcal{U}_t(c, k, n)_0$ or $\text{DCT } \mathcal{U}_t(c, k, n)_1$ respectively.
- Whichever transcoder is invoked, the results go to the same output bus via the MERGE network. Figure 14.13 shows transcoders having the same output bus width $n-1$, but in general there may be fewer outputs from $\text{DCT } \mathcal{U}_t(c, k, n)_1$ and hence fewer MERGE primitives.

A block combinator specification of the combining form proceeds in several steps. Starting with the buffer sizes

$$w = \langle \lceil \log_2 s \rceil, \lceil \log_2(c-s) \rceil \rangle \in \mathbb{N}^2 \quad (14.14)$$

a list of the two buffers $\text{DBU}^* w \in \mathbb{H}^2$ by Equation 14.10 is transformed to the block

$$(\lambda b. \mathbf{R}(b_0, b_1 \uparrow 1) \Downarrow 1) \text{DBU}^* w$$

whose inputs are the lower buffer's read request, the upper read request, the upper dual rail bus, and the lower dual rail bus in that order, with the outputs ordered similarly. The combination of the decision wait and the FORK

$$\mathbf{F}\langle \text{DW}(2, 1), \text{FORK} \rangle \uparrow 1$$

with its three outputs rotated as shown connected to the buffer block

$$\mathbf{F}_2\langle \mathbf{F}\langle \text{DW}(2, 1), \text{FORK} \rangle \uparrow 1, (\lambda b. \mathbf{R}(b_0, b_1 \uparrow 1) \Downarrow 1) \text{DBU}^* w \rangle$$

therefore has the bottom row output from the decision wait connected to upper buffer's read request. In the next step, connections from the buffer acknowledgments to the JOIN are expressed by

$$\mathbf{F}_2\langle \mathbf{F}\langle \text{DW}(2, 1), \text{FORK} \rangle \uparrow 1, (\lambda b. \mathbf{R}(b_0, b_1 \uparrow 1) \Downarrow 1) \text{DBU}^* w, \text{JOIN} \rangle$$

and the connection from the JOIN to the column input of the decision wait by

$$\mathbf{Z}(\mathbf{F}_2\langle \mathbf{F}\langle \text{DW}(2, 1), \text{FORK} \rangle \uparrow 1, (\lambda b. \mathbf{R}(b_0, b_1 \uparrow 1) \Downarrow 1) \text{DBU}^* w, \text{JOIN} \rangle \uparrow 3 \downarrow 1)$$

or more briefly by $L_0 w$ for $L_0 : \mathbb{N}^2 \rightarrow \mathbb{H}$ defined as

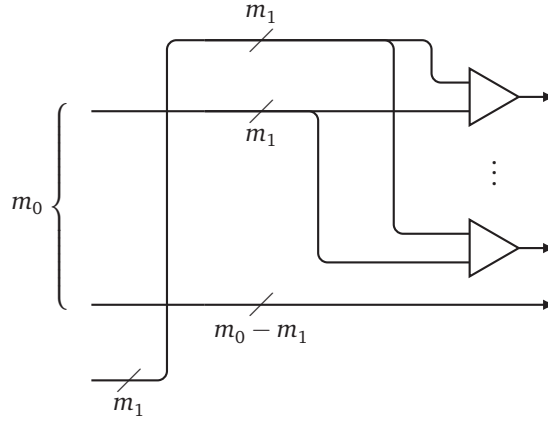
$$L_0 = \lambda w. \mathbf{Z}(\mathbf{F}_2\langle \mathbf{F}\langle \text{DW}(2, 1), \text{FORK} \rangle \uparrow 1, (\lambda b. \mathbf{R}(b_0, b_1 \uparrow 1) \Downarrow 1) \text{DBU}^* w, \text{JOIN} \rangle \uparrow 3 \downarrow 1)$$

where it should be kept in mind that the last two inputs to $L_0 w$ are connected to the decision wait rows due to the input terminal rotation, and the first output is from the FORK. Next we express the subtracter $\text{SUB}(d_0, d_0 + d_1) \in \mathbb{H}$ in terms of a list

$$d = \langle s, c-s \rangle \in \mathbb{N}^2 \quad (14.15)$$

by Equation 14.8. The subtracter's borrow lines and its output bus need to be connected to the last $2 + 2w_1$ inputs on $L_0 w$ to reach the lower buffer's input bus and the row inputs on the decision wait respectively, as in the block

$$\mathbf{L}_{2+2w_1}\langle \text{SUB}(d_0, d_0 + d_1), L_0 w \rangle$$

Figure 14.14: $\mathbf{R}(\text{MERGE}^{m_1 \leftarrow \frac{1}{2}}, \mathbf{I}^{m_0 - m_1}) \uparrow m_1$

whose expression requires no terminal rotations. The two output buses from this block are connected to the transcoders $x = \text{DCT}^* \mathcal{U}_t(c, k, n) \in \mathbb{H}^2$ in the block

$$\mathbf{L}_{2w_0+2w_1} \langle \mathbf{L}_{2+2w_1} \langle \text{SUB}(d_0, d_0 + d_1), L_0 w \rangle, (\mathcal{F} \mathbf{R}) x \rangle$$

abbreviated henceforth as $((L_1 x) L_0) d$ with $L_1 : \mathbb{H}^2 \rightarrow ((\mathbb{N}^2 \rightarrow \mathbb{H}) \rightarrow (\mathbb{N}^2 \rightarrow \mathbb{H}))$ defined as

$$L_1 = \lambda x. \lambda l. \lambda d. (\lambda w. \mathbf{L}_{2w_0+2w_1} \langle \mathbf{L}_{2+2w_1} \langle \text{SUB}(d_0, d_0 + d_1), l w \rangle, (\mathcal{F} \mathbf{R}) x \rangle) (\lambda s. \lceil \log_2 s \rceil)^* d.$$

To attend to the MERGE network, we need the list

$$m = \langle n - 1, |\bigcup \mathcal{R}((f_{k-1}^{n-1})^* \iota_{c-s})| \rangle \in \mathbb{N}^2 \quad (14.16)$$

of output bus widths according to Equation 14.13. The number of MERGE primitives is no more than m_1 , the lesser of the two, with $m_0 - m_1$ wires making up the rest of the network in a block

$$\mathbf{R}(\text{MERGE}^{m_1 \leftarrow \frac{1}{2}}, \mathbf{I}^{m_0 - m_1}) \uparrow m_1.$$

The input terminal ordering is adjusted as shown in Figure 14.14 so it slots into the block

$$\mathbf{L}_{m_0+m_1} \langle ((L_1 x) L_0) d, \mathbf{R}(\text{MERGE}^{m_1 \leftarrow \frac{1}{2}}, \mathbf{I}^{m_0 - m_1}) \uparrow m_1 \rangle$$

with the output buses from x connected appropriately to it, but the resulting block should have its output bus rotated to put the output from the FORK last.

$$\mathbf{L}_{m_0+m_1} \langle ((L_1 x) L_0) d, \mathbf{R}(\text{MERGE}^{m_1 \leftarrow \frac{1}{2}}, \mathbf{I}^{m_0 - m_1}) \uparrow m_1 \rangle \uparrow 1$$

A shorter way of expressing this block is directly in terms of the parameters determined by the decomposition

$$u = \mathcal{U}_t(c, k, n) \quad (14.17)$$

from which we have

$$m = (\lambda(s, k, n). n)^* u \quad (14.18)$$

and hence an expression $(L_2 ((L_1 x) L_0) d) u$ for $L_2 : \mathbb{H} \rightarrow (\mathbb{H}^2 \rightarrow \mathbb{H})$ given by

$$L_2 = \lambda l. ((\lambda m. \mathbf{L}_{m_0+m_1} \langle l, \mathbf{R}(\text{MERGE}^{m_1} \leftarrow \frac{1}{2}, \mathbf{I}^{m_0-m_1}) \uparrow m_1 \rangle \uparrow 1) \circ (\lambda(s, k, n). n)^*).$$

Only the FORK network shown on the front end of [Figure 14.13](#) remains to complete this construction. Letting

$$(t, v) = (2\lceil \log_2 c \rceil, 2\lceil \log_2 s \rceil)$$

denote the input bus width and the low order bus width respectively, we need v FORK primitives and $t - v$ wires for a block of the form

$$\mathbf{R}(\text{FORK}^v \mapsto \frac{1}{2}, \mathbf{I}^{t-v})$$

having t inputs and $t + v$ outputs to connect to the block $(L_2 ((L_1 x) L_0) d) u$ obtained above in

$$\mathbf{F}_{t+v} \langle \mathbf{R}(\text{FORK}^v \mapsto \frac{1}{2}, \mathbf{I}^{t-v}), (L_2 ((L_1 x) L_0) d) u \rangle$$

expressible alternatively as $L_3(c, d_0) (L_2 ((L_1 x) L_0) d) u$ in terms of $L_3 : \mathbb{N} \times \mathbb{N} \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ as defined by

$$L_3 = \lambda(c, s). \lambda l. (\lambda(t, v). \mathbf{F}_{t+v} \langle \mathbf{R}(\text{FORK}^v \mapsto \frac{1}{2}, \mathbf{I}^{t-v}), l \rangle) (2\lceil \log_2 c \rceil, 2\lceil \log_2 s \rceil).$$

However, u fully determines d according to [Equation 14.13](#), [Equation 14.15](#), and [Equation 14.17](#) as

$$d = (\lambda(s, k, n). s)^* u \quad (14.19)$$

so it would be reasonable to express the block as $L_4(c, x, u)$ with $L_4 : \mathbb{N} \times \mathbb{H}^2 \times (\mathbb{N} \times \mathbb{N} \times \mathbb{N})^2 \rightarrow \mathbb{H}$ defined as

$$L_4 = \lambda(c, x, u). (\lambda d. L_3(c, d_0) (L_2 ((L_1 x) L_0) d) u) (\lambda(s, k, n). s)^* u.$$

A definition for the combining form $\Omega_t : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{H}^2 \rightarrow \mathbb{H}$ then follows trivially as

$$\Omega_t(c, k, n, x) = L_4(c, x, \mathcal{U}_t(c, k, n))$$

thus completing the specification for dual rail to Sperner transcoders.

14.3 Sperner to dual rail conversion

Transcoding in the other direction, from Sperner to dual rail codes, amounts to implementing a function $g_k^n = (f_k^n)^{-1}$ in hardware. Further insight is possible by expressing the function as a recurrence in the same style as [Equation 14.11](#).

$$g_k^n(b) = \left(\lambda \dot{g}. \dot{g} \binom{n-1}{k} \right) \lambda s. \begin{cases} 0 & \text{if } k = n \\ g_k^{n-1}(b) & \text{if } n-1 \notin b \\ g_{k-1}^{n-1}(b - \{n-1\}) + s & \text{otherwise} \end{cases} \quad (14.20)$$

Similarly to the dual rail to Sperner transcoder, a circuit implementing g_k^n with a code size of c apparently would depend on one implementing g_k^{n-1} with a code size of s , and one implementing g_{k-1}^{n-1} with a code size of $c - s$.

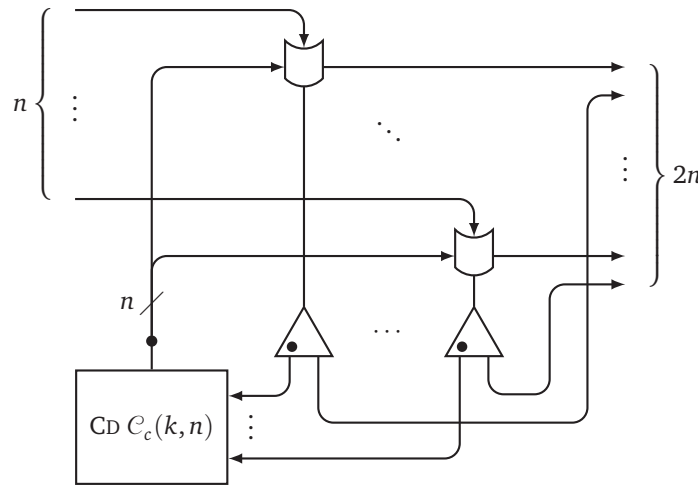


Figure 14.15: A channel expander $Cx(k, n)$ interfaces a k -of- n input channel with an n -bit dual rail output channel.

Along with a constant adder of s as described in [Section 14.1.1](#) to the partial result in the latter case, most of what is needed to complete the construction should be at hand. Nevertheless, we work up to it gradually in the remainder of this section with some preliminary considerations in [Section 14.3.1](#) and the actual derivation in [Section 14.3.2](#).

14.3.1 Preparation

Unlike the dual rail to Sperner transcoder, this one needs to test an input word b for the presence of the maximum alphabet symbol $n - 1$ instead of performing a numerical comparison. Although this test may seem like a simpler operation, there is a problem. If $n - 1$ is a member of the input word b , then a signal on the last wire in the bus is due to arrive, but if not, then no such signal is forthcoming and the circuit is obliged to detect and respond to a non-event, which is impossible for a DI circuit.

On the other hand, a completion detector designed according to [Section 13.4](#) can detect the arrival of a code word even if no particular signal is common to every word in the code. If the whole word has arrived, then every signal in it that is going to arrive has arrived, and any signal not received by then is demonstrably absent. It remains only to design a circuit that can draw this conclusion automatically.

Detecting a non-event

[Figure 14.15](#) shows a candidate. The idea is to pass each of the n bus lines carrying words of the k -of- n code $C_c(k, n)$ through the control input of a SHUNT and then through a TOGGLE before letting them reach a completion detector $CD C_c(k, n)$. (See [Equation 14.12](#).) The output from the completion detector then feeds concurrently into the data input of every SHUNT via the FORK network. There is no chance of it reaching a SHUNT ahead of a control input to the same SHUNT because it emerges

from the completion detector only after the whole word is received. Next, the data signal to each SHUNT either passes directly to its data output on the right, or is shunted downward through the alternate output of the attached TOGGLE depending on whether the SHUNT has previously received a control signal. The TOGGLE outputs shuffled together with the SHUNT data outputs form a dual rail channel with $2n$ lines, which emits a signal on exactly one line in each of the n pairs.

To return to the problem above, we can detect whether or not the symbol $n - 1$ is a member of the input word b by putting it through this circuit and monitoring the last two outputs. If the last output emits a signal, then $n - 1$ is a member of the word b , and if the penultimate output emits a signal, then $n - 1$ is not in the word b .

Expanded form

What should be done with the rest of the outputs? Keeping them around turns out to be useful for further detection and routing in the transcoder. It might even be tempting to think this circuit solves the whole problem of transcoding a k -of- n code to a dual rail code already, but it is not quite that simple. Although the output is in dual rail form, it does not encode the number $g_k^n(b) \in \mathbb{N}$ as required for compatibility with transcoding in the other direction. (One clue is that the output bus has $2n$ lines regardless of the code size c instead of $2\lceil \log_2 c \rceil$ lines.) Nevertheless, this encoding is convenient as an intermediate representation and called the **expanded form** hereafter.

Channel expanders

Specifying the circuit in Figure 14.15, called a **channel expander** hereafter, is straightforward in terms of a function $CX : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a code weight $k > 0$ and a bus width $n \geq k$ to the corresponding channel expander $CX(k, n) \in \mathbb{H}$. A cascade

$$F_n \langle \text{FORK } n, \text{SHUNT}^n \leftarrow_2^1 \text{L}_2^1, \text{TOGGLE}^n \rightarrow_2^1, \text{CD } C_c(k, n) \rangle$$

connecting the first n outputs from each stage to the first n inputs on the next takes care of connecting the FORK network to the SHUNT data inputs, the SHUNT control outputs to the TOGGLE inputs, and the dotted TOGGLE outputs to the completion detector. This block has a FORK input first and a completion detector output last, whose connection is expressible as

$$Z(F_n \langle \text{FORK } n, \text{SHUNT}^n \leftarrow_2^1 \text{L}_2^1, \text{TOGGLE}^n \rightarrow_2^1, \text{CD } C_c(k, n) \rangle \parallel 1)$$

by rolling them into position. The result has n data outputs from the SHUNT stage followed by n TOGGLE outputs. To shuffle them into the order of a dual rail channel, an output permutation $\iota_{2n} \times 2$ is needed in the definition of CX .

$$CX(k, n) = (Z(F_n \langle \text{FORK } n, \text{SHUNT}^n \leftarrow_2^1 \text{L}_2^1, \text{TOGGLE}^n \rightarrow_2^1, \text{CD } C_c(k, n) \rangle \parallel 1)) \times \iota_{2n} \times 2$$

See Equation 8.12 and Equation 8.53 for reminders about this notation.

14.3.2 Derivation

The game plan at this point is to specify a Sperner to dual rail transcoder as a cascade in two stages, with the first being a channel expander, and the second being defined by a recurrence with expanded form data in and numerical dual rail data out. The base case of the recurrence pertains to small values of n , the input bus width, or any values of n for which k the code weight is equal to n , so that

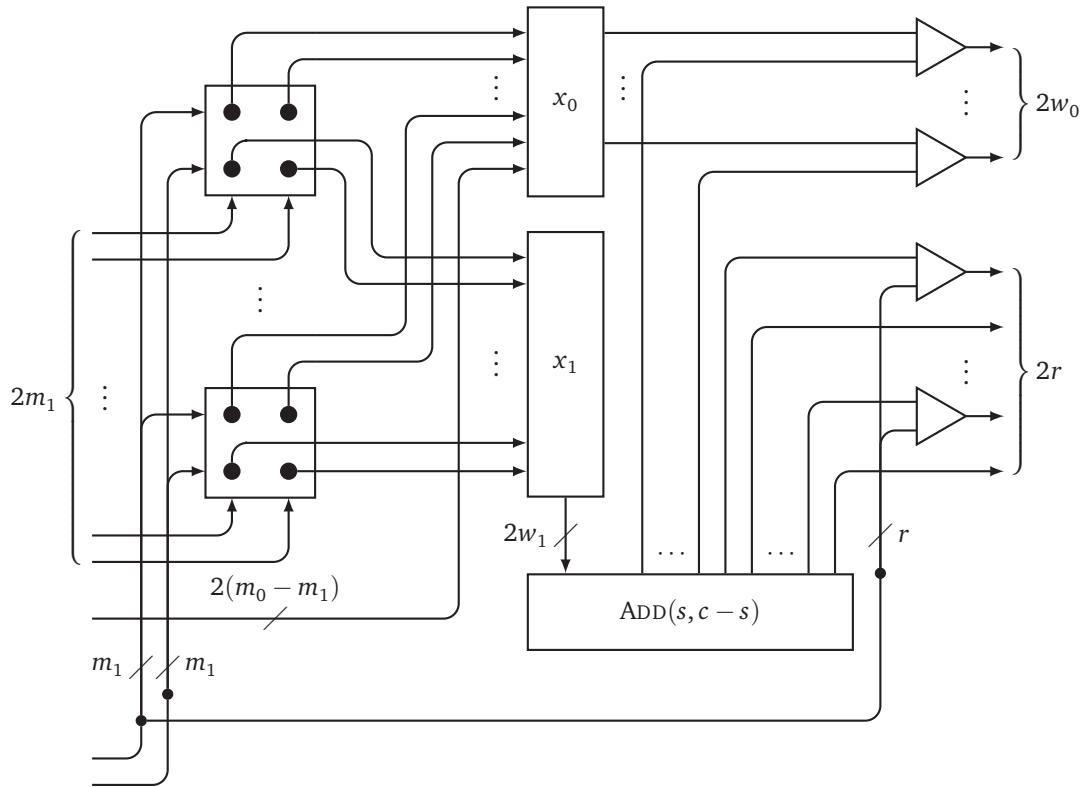


Figure 14.16: A k -of- n to dual rail transcoder for a code size c with an input bus in expanded form is recursively defined in terms of smaller versions x_0 and x_1 , where w is given by Equation 14.14, m is given by Equation 14.16, s is $(n - 1)! / (k!(n - 1 - k)!)$ and r is $\lceil \log_2 c \rceil - w_0$.

a transcoder adhering to these input and output formats is feasible to implement by Equation 13.22. The inductive case is given by a new combining form $\hat{\Omega}_t : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{H}^2 \rightarrow \mathbb{H}$ to be defined by block combinators following Figure 14.16, taking parameters c , k , n , and lower order results $x \in \mathbb{H}^2$ to the overall result $\hat{\Omega}_t(c, k, n, x) \in \mathbb{H}$.

Base case

To take the base case first, we have output words encoding numbers $a \in \mathbb{N}$ ranging from 0 to $c - 1$ inclusive in dual rail form for a code size of c . For compatibility with the dual rail to Sperner transcoder derived in Section 14.2.2, the word encoding a number a should be the set

$$\mathcal{R}((\lambda i. 2^i + (\lfloor a2^{-i} \rfloor \bmod 2))^* \iota_{\lceil \log_2 c \rceil}) \in \mathcal{P}(\mathbb{N}).$$



The compatible input word corresponding to this output would be $b = f_k^n(a) \in \mathcal{P}(\mathcal{R}(\iota_n))$ as given by Equation 14.11 were it not for the hypothesis of a front end channel expander stage mentioned above. The actual input word as far as the latter stage of the transcoder is concerned is

a set of $2k$ symbols in the range of 0 to $2n - 1$ inclusive. For each member i of the external word b , there is a symbol $2i + 1$ in the internal expanded form, and for each non-member $i \in \mathcal{R}(\iota_n) - b$, there is a symbol $2i$ in the expanded form. It would be more succinct therefore to describe the input word as $h_k^n(a)$ in terms of a function $h_k^n : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$ defined as follows.

$$h_k^n = (\lambda b. (\mu \lambda i. 2i + \delta_b^{b \cup \{i\}}) \mathcal{R}(\iota_n)) \circ f_k^n$$

A transcoder specification by Equation 13.22 is parameterized by a set of pairs of input words and their corresponding outputs for the whole code.

$$\text{Tc } \mathcal{R}((\lambda a. (h_k^n(a), \mathcal{R}((\lambda i. 2^i + (\lfloor a2^{-i} \rfloor \bmod 2))^* \iota_{\lceil \log_2 c \rceil})))^* \iota_c)$$

This expression serves as the base case of a recurrence to be defined presently.

Inductive case

For the inductive case, we need to look at Figure 14.16 more carefully and find an expression for a combining form $\hat{\Omega}_t(c, k, n, x)$ that captures it in writing, but first a walk through is in order.

- An input word in expanded form enters from the left, with the highest numbered lines $2n - 2$ and $2n - 1$ shown at the bottom left.
- A signal to the bottom input line indicates that $n - 1$ is a member of the input word b (with respect to Equation 14.20), so it is broadcast by a FORK network to the bottom row inputs of the decision waits, thereby routing all other signals on the bus to x_1 .
- By hypothesis, x_1 generates the dual rail representation of $b - \{n - 1\}$, and transmits it to an adder of s , which sends the sum to the output bus via the back end MERGE networks.
- A signal to the penultimate input indicates that $n - 1$ is not a member of the input word, so it is broadcast to the top row inputs of the decision waits to route the rest of the signals to x_0 .
- In the case depicted, x_0 has r fewer output bits than the adder, so this signal must also be broadcast to the r most significant zero bits on the output bus to make up the full word via the lower part of the back end MERGE network.
- By hypothesis, x_0 generates the dual rail representation of b because it does not contain $n - 1$, which is transmitted to the output via the upper part of the back end MERGE network.

A good place to start on this construction is with the block

$$\mathbf{F}_{2w_0} \langle x_0, \text{MERGE}^{2w_0 \leftarrow 1} \rangle$$

consisting of x_0 and the upper part of the back end MERGE network by continuing to denote the widths w according to Equation 14.14, so that w_0 is $\lceil \log_2 s \rceil$. Similarly, the block

$$\mathbf{C}_{2w_1} \langle x_1, \text{ADD}(d_0, d_1) \rangle$$

covers x_1 and the adder based on $d = \langle s, c - s \rangle$ as in Equation 14.15. Connections from the first $2w_0$ outputs of the adder to the remaining inputs on the upper MERGE network are expressible as

$$\mathbf{D}_{2w_0} \langle \mathbf{C}_{2w_1} \langle x_1, \text{ADD}(d_0, d_1) \rangle, \mathbf{F}_{2w_0} \langle x_0, \text{MERGE}^{2w_0 \leftarrow 1} \rangle \rangle$$

and summarized as $L_5(d, x)$ in terms of a function $L_5 : \mathbb{N}^2 \times \mathbb{H}^2$ defined by

$$L_5 = \lambda(d, x). (\lambda w. \mathbf{D}_{2w_0} \langle \mathbf{C}_{2w_1} \langle x_1, \text{ADD}(d_0, d_1) \rangle, \mathbf{F}_{2w_0} \langle x_0, \text{MERGE}^{2w_0} \leftarrow_2^1 \rangle \rangle) (\lambda s. \lceil \log_2 s \rceil)^* d$$

taking advantage of the relationship between w and d . If we continue to denote the input widths $m \in \mathbb{N}^2$ according to [Equation 14.16](#) and let

$$r = \lceil \log_2 c \rceil - \lceil \log_2 s \rceil = \lceil \log_2 c \rceil - \lceil \log_2 d_0 \rceil = \lceil \log_2 c \rceil - w_0 \quad (14.21)$$

this definition makes $L_5(d, x)$ a block with $2m_1$ inputs to x_1 followed by $2m_0$ inputs to x_0 , and $2r$ left over outputs from the adder (which may be none) followed by $2w_0$ outputs from the MERGE network. Making a mental note that the inputs to x_1 come before those of x_0 , we move on to the array of m_1 decision waits

$$\text{DW}(2, 2)^{m_1} \leftarrow_4^2 \rightarrow_4^2$$

with its inputs and outputs grouped into four buses of width $2m_1$ each, and the pair of FORK networks in parallel

$$(\mathcal{F} \mathbf{R}) \text{FORK}^* \langle m_1 + 1 - \delta_0^r, m_1 \rangle$$

whose first needs an extra output to broadcast to the lower back end MERGE network only if r is non-zero. Shuffling the last $2m_1$ FORK outputs

$$(\mathcal{F} \mathbf{R}) \text{FORK}^* \langle m_1 + 1 - \delta_0^r, m_1 \rangle \times \iota_{2m_1} \times 2$$

enables each FORK network to reach one row on each decision wait in the block

$$\mathbf{U}_{2m_1} \langle (\mathcal{F} \mathbf{R}) \text{FORK}^* \langle m_1 + 1 - \delta_0^r, m_1 \rangle \times \iota_{2m_1} \times 2, \text{DW}(2, 2)^{m_1} \leftarrow_4^2 \rightarrow_4^2 \rangle$$

whose last $4m_1$ outputs would be better to permute by

$$\iota_{2m_1}^{2m_1} \parallel \iota_{2m_1}$$

thereby interchanging the first and second bundles of $2m_1$ bus lines from the decision waits in preparation to connect them to $L_5(d, x)$ in that order. Let this block be denoted $L_6(m, r)$ in terms of a function $L_6 : \mathbb{N}^2 \times \mathbb{N} \rightarrow \mathbb{H}$ given by

$$L_6 = \lambda(m, r). \mathbf{U}_{2m_1} \langle (\mathcal{F} \mathbf{R}) \text{FORK}^* \langle m_1 + 1 - \delta_0^r, m_1 \rangle \times \iota_{2m_1} \times 2, \text{DW}(2, 2)^{m_1} \leftarrow_4^2 \rightarrow_4^2 \rangle \times \iota_{m_1}^{m_1} \parallel \iota_{m_1}$$

where it should be noted that the two FORK network inputs come before the $2m_1$ decision wait column inputs and therefore have to be rolled to the end at some point, but not yet, and the extra FORK output for the back end, if any, precedes the $4m_1$ decision wait outputs. Before rolling the FORK inputs, we first need to construct the block

$$\mathbf{U}_{4m_1} \langle L_6(m, r), L_5(d, x) \rangle$$

whose first inputs are to the FORK networks, whose next $2m_1$ inputs are to the decision waits, and whose last $2(m_0 - m_1)$ outputs are those of x_0 , so that writing

$$l = \mathbf{U}_{4m_1} \langle L_6(m, r), L_5(d, x) \rangle \uparrow 2 \quad (14.22)$$

would make l a network whose inputs have the order depicted in [Figure 14.16](#), whose first output is from a FORK to the back end if r is non-zero, whose next $2r$ outputs are from the adder, and whose

last $2w_0$ outputs are from the upper back end MERGE network. If r is zero, then l is already the whole result, but otherwise it needs an additional network of the form

$$\mathbf{F}_r \langle \text{FORK } r, \mathbf{R}(\text{MERGE}, \mathbf{I})^r \leftarrow \mathbb{1}_3 \rangle$$

representing a dual rail channel of r bits with an extra FORK input at the beginning that can cause it to emit a word of zeros. Connecting l to this network is simple

$$\mathbf{F}_{2r+1} \langle l, \mathbf{F}_r \langle \text{FORK } r, \mathbf{R}(\text{MERGE}, \mathbf{I})^r \leftarrow \mathbb{1}_3 \rangle \rangle$$

and allowing for the possibility of $r = 0$ likewise.

$$\langle \mathbf{F}_{2r+1} \langle l, \mathbf{F}_r \langle \text{FORK } r, \mathbf{R}(\text{MERGE}, \mathbf{I})^r \leftarrow \mathbb{1}_3 \rangle \rangle, l \rangle_{\delta_0^r}$$

Letting this expression be denoted $L_7(l, r)$ in terms of a function $L_7 : \mathbb{H} \times \mathbb{N} \rightarrow \mathbb{H}$ defined by

$$L_7 = \lambda(l, r). \langle \mathbf{F}_{2r+1} \langle l, \mathbf{F}_r \langle \text{FORK } r, \mathbf{R}(\text{MERGE}, \mathbf{I})^r \leftarrow \mathbb{1}_3 \rangle \rangle, l \rangle_{\delta_0^r}$$

we can summarize the network as $L_8(c, d, m, x)$ in terms of a function $L_8 : \mathbb{N} \times \mathbb{N}^2 \times \mathbb{N}^2 \times \mathbb{H}^2 \rightarrow \mathbb{H}$ defined by

$$L_8 = \lambda(c, d, m, x). (\lambda r. L_7(\mathbf{U}_{4m_1} \langle L_6(m, r), L_5(d, x) \rangle \uparrow 2, r)) [\log_2 c] - [\log_2 d_0]$$

using Equation 14.21 and Equation 14.22. The desired combining form $\dot{\Omega}_t$ to capture Figure 14.16 then follows directly in terms of L_8 , Equation 14.17, Equation 14.18, Equation 14.19, and the decomposition function \mathcal{U}_t as defined by Equation 14.13.

$$\dot{\Omega}_t(c, k, n, x) = (\lambda u. L_8(c, (\lambda(s, k', n'). s)^* u, (\lambda(s, k', n'). n')^* u, x)) \mathcal{U}_t(c, k, n)$$

Recurrence

Now that both the base and inductive cases are derived, we can put them together into a function $L_9 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ defined by a recurrence.

$$L_9(c, k, n) = \begin{cases} \text{Tc } \mathcal{R}((\lambda a. (h_k^n(a), \mathcal{R}((\lambda i. 2^i + [\lfloor a 2^{-i} \rfloor \bmod 2])^* \iota_{\lfloor \log_2 c \rfloor})))^* \iota_c) & \text{if } (k, n) \in K_t \\ \dot{\Omega}_t(c, k, n, L_9^* \mathcal{U}_t(c, k, n)) & \text{otherwise} \end{cases}$$

The set $K_t \subseteq \mathbb{N} \times \mathbb{N}$ specifies the values of k and n for which the transcoder is computed the hard way by Equation 13.22. At a minimum, it must contain all pairs of the form $(1, n)$ and $(n, n) \in \mathbb{N} \times \mathbb{N}$, and can include any others deemed advantageous for implementation reasons.

This result is not the whole transcoder because it relies on the input being in expanded form as discussed in Section 14.3.1. The construction of a Sperner code to dual rail transcoder $\text{CDT}(c, k, n)$ is complete in combination with a channel expander as a front end $\text{Cx}(k, n)$.

$$\text{CDT}(c, k, n) = \mathbf{C}_{2n} \langle \text{Cx}(k, n), L_9(c, k, n) \rangle \quad (14.23)$$

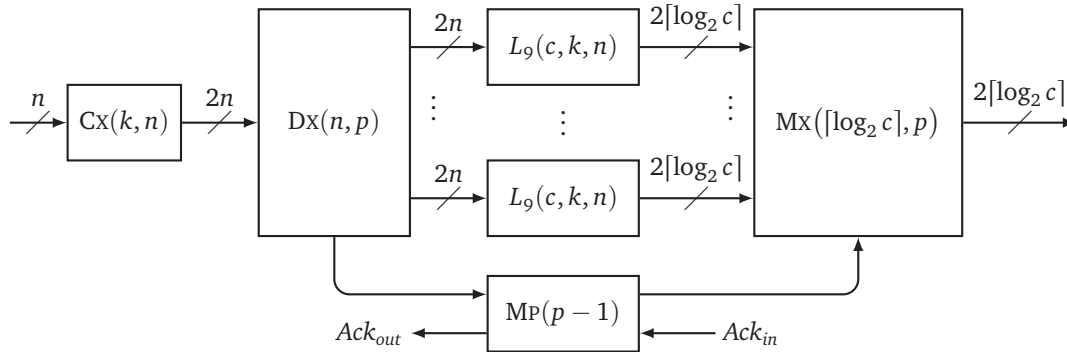
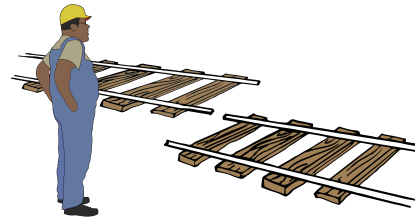


Figure 14.17: A parallel handshaking Sperner to dual rail transcoder transcodes up to p words concurrently.

14.4 Parallelism

As inevitable as these transcoder designs may seem in retrospect, they are not a fitting conclusion to this chapter if the latency incurred by the conversion from one format to the other negates any other advantages they may have. In this section we investigate a way of speeding them up if necessary by throwing more hardware at them. The general idea is to put a group of transcoders in parallel, farm each incoming word out to a separate one, and then funnel all of their results into a single channel on the other side. The number of parallel transcoders can be chosen so that in steady state operation, the longest running transcoder in the group is ready for a new input word by the time the rest are loaded, and the group of transcoders working together therefore can dispatch the input words as fast as the channel can deliver them.



More easily said than done, this arrangement entails a bit of overhead in the way of the routing and recombining stages, as well as some way of regulating the throughput to prevent overflow, but is well worth developing because it exemplifies a pattern that is broadly applicable to tackling many parallel computation problems other than transcoding. The rest of this section is organized around elaborating on the hitherto undefined blocks in Figure 14.17, which pertains specifically to the example of Sperner to dual rail transcoders, discussing their operation, and deriving an expression for the result overall.

14.4.1 Dual rail toggles

A first step along these lines is the design of a network that can route a dual rail channel successively to each of p destinations in a cyclic sequence. To make it simpler, we can start by considering the case of a 1-bit dual rail channel with each word routed alternately to one of two destinations, which reduces to the circuit shown in Figure 14.18. The TOGGLE triggered by each incoming bit, whatever it may be, alternately selects either the top or the bottom row on the decision wait, thereby routing the bit to one output channel or the other accordingly. This circuit is expressible as a one-liner

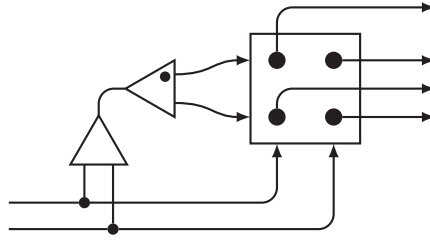


Figure 14.18: a dual rail toggle cell, $DT \in \mathbb{H}$

$DT \in \mathbb{H}$ with not much further explanation needed.

$$DT = C_4 \langle F_2 \langle FORK^2 \uparrow_2^1, C \langle MERGE, TOGGLE \rangle \rangle \uparrow 2, DW(2, 2) \rangle \quad (14.24)$$

The next step is to generalize this circuit from two output channels to any $p > 0$ output channels, while still restricting attention to channels carrying one bit each. For this purpose, let $DT : \mathbb{N} \rightarrow \mathbb{H}$ denote a function taking a number of channels $p \in \mathbb{N}$ to a dual rail toggle $DT(p) \in \mathbb{H}$ with a 1-bit input channel and p 1-bit output channels.

The case of $p = 1$ is trivially a pair of wires l^2 , but otherwise it is appropriate define the function as a recurrence in two cases depending on whether p is odd or even, following patterns similar to those shown in Figure 9.16 and Figure 9.17 for a generalized TOGGLE primitive.

- For even values of p , the result takes the form of a balanced tree rooted by the dual rail toggle cell DT defined by Equation 14.24 and connected to a parallel combination of two of multi-way dual rail toggles $(DT \ p/2)^2$ each half the size in a block $C_4 \langle DT, (DT \ p/2)^2 \rangle$. The necessary output permutation to put the output buses in the right order is similar to $\iota_n \times 2$ as shown in Equation 9.22, except that it pertains to p pairs of bus lines instead of n bus lines. A simple adjustment to $b(\lambda i. \iota_2^{2i})^* (\iota_p \times 2)$ has the effect of expanding each line number to a pair of consecutive line numbers.
- For odd values of p , we take a cue from Figure 9.17 by generating $DT \ p + 1$ and feeding one of the output channels back to the input. A parallel combination $MERGE^2$ connected to the toggle by $C_2 \langle MERGE^2, DT \ p + 1 \rangle$ with the MERGE inputs shuffled in $C_2 \langle MERGE^2, DT \ p + 1 \rangle \leftarrow_2^1$ allows for the necessary two-line feedback path in $Z^2(C_2 \langle MERGE^2, DT \ p + 1 \rangle \leftarrow_2^1)$.

Putting these cases together leads to the following definition.

$$DT(p) = \begin{cases} \langle Z^2(C_2 \langle MERGE^2, DT \ p + 1 \rangle \leftarrow_2^1), l^2 \rangle_{\delta_1^p} & \text{if } p \bmod 2 = 1 \\ C_4 \langle DT, (DT \ p/2)^2 \rangle \times b(\lambda i. \iota_2^{2i})^* (\iota_p \times 2) & \text{otherwise} \end{cases}$$

14.4.2 Channel demultiplexers

A multi-way toggle for an n -bit dual rail channel would be easy to obtain by putting n single bit multi-way toggles together in parallel,

$$(DT \ p)^n \times b(\lambda i. \iota_2^{2i})^* \iota_{pn} \times n$$

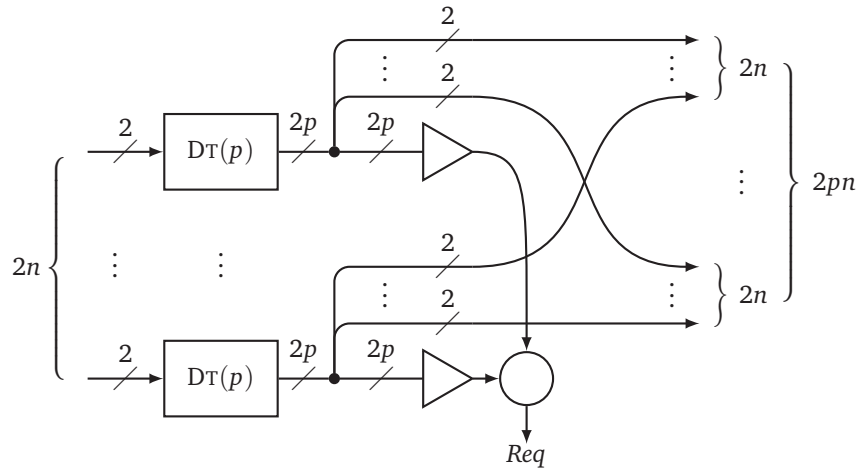


Figure 14.19: a dual rail channel demultiplexer $DX(n, p)$ for n bits and p channels

and permuting the outputs as shown to group them into p buses of $2n$ lines each. However, using it effectively as part of the routing stage in Figure 14.17 requires it to have an additional completion detection signal as an output. The completion detection signal should indicate that an input word has worked its way through the decision waits in the toggles so that it is safe for the sender to transmit another one, at least as far as the routing stage is concerned.

Something like what is shown in Figure 14.19 would do the trick, which is called a dual rail channel demultiplexer for the sake of this discussion. In addition to the array of multi-way toggles, it features a network $(MERGE\ 2p)^n$ synchronized by $C_n \langle (MERGE\ 2p)^n, JOIN\ n \rangle$ to generate the completion detection signal. The inputs to the MERGE network are taken from the latter $2pn$ outputs of a network $FORK^{2pn} \Gamma_2^1$, all of which form a cascade

$$L_{2pn} \langle (DT\ p)^n, FORK^{2pn} \Gamma_2^1, C_n \langle (MERGE\ 2p)^n, JOIN\ n \rangle \rangle$$

in combination with the toggle array. This expression describes a network whose completion detection signal is the last output, and whose other outputs still need to be grouped into p buses of $2n$ lines each as discussed above. Hence we define

$$DX(n, p) = L_{2pn} \langle (DT\ p)^n, FORK^{2pn} \Gamma_2^1, C_n \langle (MERGE\ 2p)^n, JOIN\ n \rangle \rangle \downarrow 1 \times b(\lambda i. \iota_2^{2i})^* \iota_{pn} \times n$$

as a function $DX : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a number of bits n and a number of channels p to a block $DX(n, p) \in \mathbb{H}$ that demultiplexes an n -bit dual rail channel over p dual rail channels of n bits each, and whose first output is a completion detection signal.

14.4.3 Channel multiplexers

The back end of the parallel transcoder shown in Figure 14.17, called a dual rail channel multiplexer for this discussion, does the opposite of the demultiplexer by merging several channels into one. Because each input channel carries a word from one of the blocks labeled $L_9(c, k, n)$ in the middle,

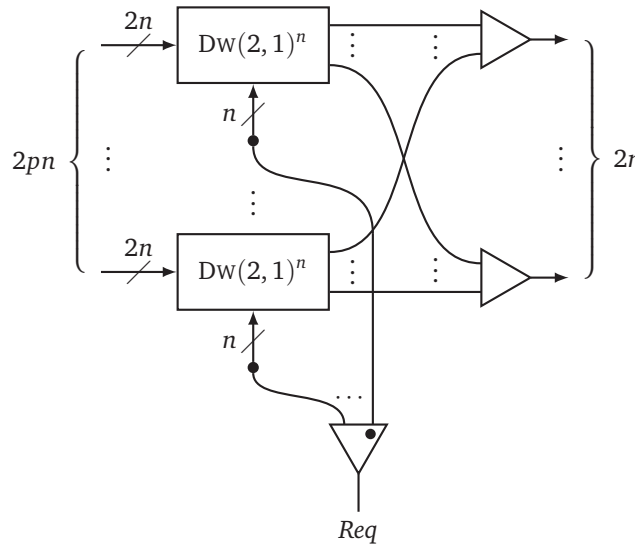


Figure 14.20: a dual rail channel multiplexer $Mx(n, p)$ for n bits and p channels

there could be more than one of them available at the same time, but the multiplexer must send no more than one at a time to the output channel. To avoid sending them too often, the multiplexer sends a word only in response to a request signal, which we envision for the moment to be under the control of whatever is on the receiving end of the output channel. Therefore the multiplexer needs some way of buffering the input words it is waiting to send, and some way of selecting the next one to send.

Buffering

The dual rail buffer developed in Section 14.1.3 is overkill for this situation because the acknowledgment and overwriting functionality are unnecessary. A simple array of the form $DW(2, 1)^n$ does the job of storing one word from an n -bit dual rail channel, and an array $DW(2, 1)^{pn}$ can store a word concurrently from up to p channels of n bits each. When combined with a FORK network in

$$L_{pn} \langle \langle \text{FORK } n \rangle^p, DW(2, 1)^{pn \leftarrow 2} \rangle$$

it becomes a block with p selection inputs followed by p buses of $2n$ lines each, and output buses similarly organized. A signal to the i -th selection input causes dual rail data written to the i -th input bus to be forwarded to the i -th output bus, either immediately if it has already been written or subsequently after it is.

Selection

As a matter of maintaining data integrity, the multiplexer must transmit the output words in an order corresponding to the order in which their antecedents are received by the transcoder. Because the demultiplexer sends one word to each block labeled $L_9(c, k, n)$ in sequence starting with the

first, the multiplexer must select data from its input buses in the same order. A TOGGLE network with p outputs connected to the selection inputs on the FORK network keeps track of this sequence naturally.

$$\mathbf{F}_p \langle \text{TOGGLE } p, \mathbf{L}_{pn} \langle (\text{FORK } n)^p, \text{DW}(2, 1)^{pn \leftarrow 3} \rangle \rangle$$

This expression describes a block with one request input followed by $2pn$ data inputs and $2pn$ data outputs organized into p buses of $2n$ lines each.

Merging

The rest of the multiplexer only needs to merge the outputs from the block described above. An array $(\text{MERGE } p)^{2n}$ has the right number of inputs, $2pn$, and an input permutation network given by $\iota_{2pn} \times p$ would let each block $\text{MERGE } p$ in the array take an input from one of the p buses. Hence we can complete the specification of the channel multiplexer as shown in Figure 14.20 by writing

$$\text{MX}(n, p) = \mathbf{C}_{2pn} \langle \mathbf{F}_p \langle \text{TOGGLE } p, \mathbf{L}_{pn} \langle (\text{FORK } n)^p, \text{DW}(2, 1)^{pn \leftarrow 3} \rangle, \iota_{2pn} \times p \times (\text{MERGE } p)^{2n} \rangle \quad (14.25)$$

to define a function $\text{MX} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a number of bits n and a number of channels p to the channel multiplexer $\text{MX}(n, p) \in \mathbb{H}$.

14.4.4 Micropipeline controllers

The last piece of the puzzle in Figure 14.17 is the mysterious block labeled $\text{MP}(p - 1)$, which must interface the request signals from the multiplexer and demultiplexer stages with exposed acknowledgments in and out such that the parallel transcoder acts as an intermediary between a k -of- n code producer on the left and a dual rail code consumer on the right (not shown). The acknowledgment out tells the producer to send another k -of- n word to the transcoder, and the acknowledgment in tells the transcoder to send another dual rail word the consumer.



One way this block could work is just by passing the buck. The request output from the demultiplexer connected directly to the request input of the multiplexer would make the latter forward a word to the consumer whenever the demultiplexer receives one from the producer, and an acknowledgment in wired directly to the acknowledgment out would inform the producer of the need for another word when the consumer is ready. This protocol would be correct but self-defeating, because there would never be more than one of the blocks labeled $L_9(c, k, n)$ active at once, and hence no performance advantage over the design given by Equation 14.23.

A more sophisticated alternative would be for the block to acknowledge the first $p - 1$ requests without waiting for an acknowledgment from the consumer just to get all p of the transcoder blocks up and running as soon as possible. Then it could send one request to the multiplexer to start the cycle, and then pass the buck as above. This alternative would enable concurrent transcoding but would entail a permanent backlog of at least $p - 1$ words buffered by the multiplexer stage that never move until the producer sends more, even if the consumer is ready to receive them. Deadlock would be a possibility if the producer were dependent for any reason on the progress of the consumer.

If neither of these ideas is acceptable, maybe the block needs to implement some sort of a counter to keep track of the number of words currently being transcoded, which can be in the

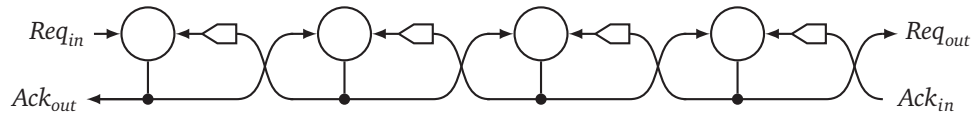


Figure 14.21: A micropipeline controller has a passive 2Φ port on the left, an active 2Φ port on the right, and p stages between them, shown here for $p = 4$. It can acknowledge up to p requests from the left before waiting for an acknowledgment from the right.

range of 0 to p . The counter would have to be able to count up or down, with incoming requests incrementing and incoming acknowledgments decrementing the count (presumably with arbitration to deal with simultaneous inputs), such that it withholds acknowledgments out when the count hits p and withholds requests out when the count hits 0, but otherwise emits them concurrently according to a 2Φ protocol on each port whenever the count permits. To judge by the complexity of the adders and subtracters developed in [Section 14.1.1](#) and [Section 14.1.2](#), this block is starting to look like quite a complicated proposition.

It is all the more striking therefore that a design proposed in [273] called a *micropipeline controller* solves this problem with such simplicity. The closest thing to poetry in circuit design, it consists of a cascade of stages each composed of only three primitives as shown in [Figure 14.21](#). It has a passive 2Φ port shown at the left and an active one shown at the right, and can have any number of stages including zero, in which case it reduces to a pair of wires ¹². If attempts to visualize its steady state operation evoke the imagery of a standing wave, then maybe some insight can come from experimenting mentally with the boundary conditions.

- Regardless of the number of stages, the first request in from the left propagates to the request out on the right without any blocking.
- If there is at least one stage, then the first request input is also reflected immediately back to the left by the first stage without any acknowledgment needed from the right.
- If an acknowledgment comes from the right in response to the first outgoing request, and there have been no more incoming requests, it propagates no further than the last stage (if there is one), but blocks nothing because the initial request has already been acknowledged.
- Alternatively, up to p incoming requests can be acknowledged initially without any acknowledgment from the right, where p is the number of stages, but no more than one request emerges on the right until after the first acknowledgment is received there.
- After the first p requests and acknowledgments on the left with no acknowledgment from the right, the 2Φ protocol allows another incoming request of course, but its acknowledgment is blocked until at least one incoming acknowledgment arrives from the right.
- If a backlog of p incoming requests accumulates, each incoming acknowledgment from the right lets out another one until they are used up, with no further requests needed from the left. However, the first incoming acknowledgment in this case propagates all the way back to the first stage, thereby enabling the previously blocked acknowledgment out and inviting another request in.

In general, both ports can interact with the environment concurrently without interruption provided neither of them ever gets more than $p + 1/2$ handshakes ahead of the other.

A formal specification of a micropipeline controller is straightforward as a function $MP : \mathbb{N} \rightarrow \mathbb{H}$ taking a number of stages $p \in \mathbb{N}$ to a micropipeline controller $MP(p) \in \mathbb{H}$ having p stages. The first input of a single stage

$$F\langle \text{PUSH, JOIN, FORK} \rangle$$

is the acknowledgment and the second input is the request. The outputs from the FORK are interchangeable but as a matter of convention the first output is designated as the acknowledgment out and the second as the request out. To connect a new first stage to a micropipeline controller t , we first need to connect the request out from the first stage to the request in on t

$$L\langle F\langle \text{PUSH, JOIN, FORK} \rangle, t \rangle$$

and then connect the acknowledgment out from t , which is the second output on this block, to the acknowledgment in on the first stage, which is the first input. Hence we roll both the inputs and the outputs up and then connect the first output to the last input,

$$Z(L\langle F\langle \text{PUSH, JOIN, FORK} \rangle, t \rangle \uparrow 1)$$

resulting in a new micropipeline controller with one more stage than t but with the terminals interchanged. To maintain the convention of acknowledgments first, we have to roll it one more time.

$$(Z(L\langle F\langle \text{PUSH, JOIN, FORK} \rangle, t \rangle \uparrow 1)) \Downarrow 1$$

A micropipeline controller with p stages is obtained by applying this operation p times to one with zero stages, suggesting the following definition for $MP : \mathbb{N} \rightarrow \mathbb{H}$

$$MP(p) = (\varepsilon_p \lambda(h, t). (Z(L\langle F\langle \text{PUSH, JOIN, FORK} \rangle, t \rangle \uparrow 1)) \Downarrow 1) \iota_p$$

as a fold over ι_p with vacuous case I^2 .

14.4.5 A parallel transcoder

The rest of this section is about putting everything in [Figure 14.17](#) together. A combination of the channel expander and the demultiplexer stages in a parallel constant weight to dual rail transcoder is expressible as

$$C_{2n}\langle CX(k, n), DX(n, p) \rangle$$

where n is the input bus width, k is the code weight, and p is the number of blocks labeled $L_9(c, k, n)$ in the figure. The transcoder needs a micropipeline controller $MP(p - 1)$ with one less stage than the number of these blocks so that the producer sends at most p words until at least one of them is consumed. For example, if there were only one of them, then only a degenerate micropipeline controller $MP(0) = I^2$ would be appropriate. Recalling that the request line is the first output from the demultiplexer and the last input to the micropipeline controller, we could combine them in an expression of the form

$$ZR(L_{2pn}\langle C_{2n}\langle CX(k, n), DX(n, p) \rangle, MP(p - 1) \rangle)$$

but it would be better to rotate the micropipeline controller outputs putting the request output ahead of the acknowledgment for reasons to become apparent shortly,

$$ZR(L_{2pn}\langle C_{2n}\langle CX(k, n), DX(n, p) \rangle, MP(p - 1) \downarrow 1 \rangle)$$

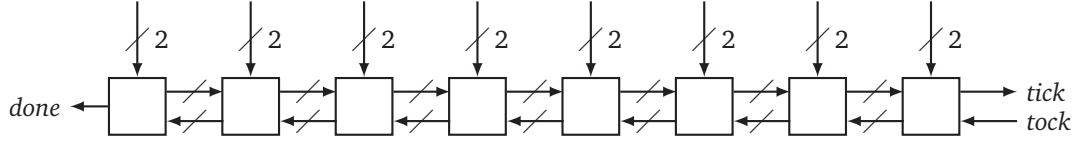


Figure 14.22: An eight bit loadable counter takes a number from 0 to 255 in dual rail form from above, initiates that number of handshakes on the right, and then sends a signal to the left.

and to abbreviate this expression as $l_0(k, n, p)$ with $l_0 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ defined as follows

$$l_0 = \lambda(k, n, p). \mathbf{ZR}(\mathbf{L}_{2pn} \langle \mathbf{C}_{2n} \langle \mathbf{CX}(k, n), \mathbf{DX}(n, p) \rangle, \mathbf{MP}(p-1) \downarrow 1 \rangle)$$

so that its first $2pn$ outputs are dual rail buses, its next output is a request from the micropipeline controller, and its last is an acknowledgment from the micropipeline controller. We also note an input bus width of $2pm$ for the multiplexer stage, where $m = \lceil \log_2 c \rceil$ depends on the output bus widths of the blocks $l = L_9(c, k, n)$, which depend on the code size c , but this bus comes only after the first input to the multiplexer, which is the request line, based on Equation 14.25. These blocks fit together therefore in a block $l_1(c, p) L_9(c, k, n)$ with $l_1 : (\mathbb{N} \times \mathbb{N}) \rightarrow (\mathbb{H} \rightarrow \mathbb{H})$ defined by

$$l_1 = \lambda(c, p). \lambda l. (\lambda m. \mathbf{L}_{2pm} \langle l^p, \mathbf{MX}(m, p) \rangle) \lceil \log_2 c \rceil$$

whose first $2pn$ inputs carry dual rail data and whose last is the request input to the multiplexer. To specify the whole transcoder, it remains only to define

$$\mathbf{PCDT}(c, k, n, p) = \mathbf{F}_{2pn+1} \langle l_0(k, n, p), l_1(c, p) L_9(c, k, n) \rangle$$

as a function $\mathbf{PCDT} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{H}$ taking a code size c , a code weight k , an input bus width n and a degree of parallelism p to a parallel constant weight to dual rail transcoder $\mathbf{PCDT}(c, k, n, p) \in \mathbb{H}$ with an input bus of width n followed by an acknowledgment input, and an acknowledgment output followed by an output bus of width $2\lceil \log_2 c \rceil$.

Pipe dreams

1. What would have to be different about the adder and subtracter cells shown in [Figure 14.1](#) if they took two operands x and y instead of just x and a hard wired constant?
2. Is there a cheaper implementation of the full borrow propagation network cells in [Figure 14.7](#) that avoids using a decision wait? Is there one for the dual rail toggle in [Figure 14.18](#)?
3. What asymptotic latency should we expect from a channel expander ([Figure 14.15](#)) in terms of k and n ? Is there any way to speed it up by parallelizing it?
4. Would it be a bigger mistake to make the micropipeline controller too short for the rest of the parallel transcoder, or too long? Should we infer that shorter micropipeline controllers refine longer ones, or *vice versa*? What is the answer according to [Equation 8.34](#)?
5. The original application proposed in [273] for a micropipeline controller pertained to data paths as elastic pipelines. How could it be modified for use in combination with an array of decision waits to make a buffered dual rail channel, and when would such a thing be useful? Is a longer pipeline always a more capable replacement for a shorter one in that case?
6. Design a parallel dual rail to Sperner transcoder following the pattern of [Section 14.4](#). (hint: It may be convenient to use expanded form data as an intermediate representation and something like the opposite of a channel expander on the back end.)
7. [Figure 14.22](#) shows a half baked idea for a loadable counter, which takes a number in dual rail form, engages in that number of handshakes on an active 2Φ port, and then issues a separate acknowledgment. Design the circuits that belong inside the cells. Use any number of wires between cells but connect them only to their nearest neighbors. (hint: Start with a verbal description of the protocol each cell observes.)
8. Is a generalization of the transcoder design in [Figure 14.16](#) involving multidimensional decision waits possible, and if so, what advantage could it have?
9. Suppose the time to transcode a word varies greatly, but transmitting the words out of order is acceptable. Design load balancing versions of the channel multiplexer and demultiplexer stages in [Figure 14.17](#) for better performance. (hint: Use arbiters.)
10. Design an arbiter that grants up to a constant number p of concurrent requests, reducing to an ordinary arbiter for $p = 1$. (hint: Use micropipeline controllers.)
11. How about a token ring arbiter made from a busy-waiting circular micropipeline controller and no more ARB primitives ever: genius or madness?

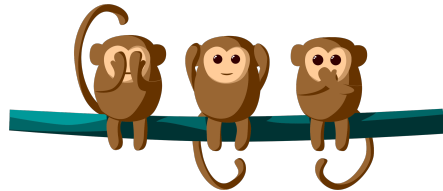


Part IV

Synthesis

STATE BASED SYNTHESIS

The theme of circuit synthesis occupying the rest of the book pertains literally to synthesizing circuits automatically from behavioral descriptions, but also entails a synthesis of concepts from previous chapters within a common context. From Petri nets and graph algorithms to transducers and refinement, it happens that decision waits, arbiters, transcoders, and some minor players all have a role before this chapter is through. However, the motivation is not one of yet more idle theoretical inquiry. Quite the contrary, the end game should be to forget the theory and get the job done, but as a start we can try forgetting about everything but process combinators as introduced in [Chapter 5](#), netlists as defined in [Chapter 8](#), and an algorithm for obtaining one from the other that suppresses the details.



Constructing such an algorithm unfortunately requires at least some attention to its inner workings. The specific approach called state based synthesis pursued in this chapter depends on the transducer model $\mathbf{T}(X)$ proposed in [Chapter 7](#) as an intermediate representation between a source Petri net modeled specification $X \in \mathbb{D}$ and a target hierarchical block representation $\mathcal{F}_{\mathbb{D}\mathbb{H}}^\alpha(X) \in \mathbb{H}$ in terms of a transformation $\mathcal{F}_{\mathbb{D}\mathbb{H}}^\alpha : \mathbb{D} \rightarrow \mathbb{H}$ to be developed presently through a sequence of incremental improvements. To go end to end from a process level description to a netlist, we would begin with process combinators as noted above, which directly yield a member of \mathbb{D} , and finish with the transformation $\mathcal{F}_{\mathbb{H}\mathbb{L}} : \mathbb{H} \rightarrow \mathbb{L}$ as defined by [Equation 8.23](#) to obtain the netlist. These two phases are taken for granted hereafter.

A word about the limitations of this approach is in order. The most severe limitation is that the need for a transducer as an intermediate representation restricts this approach to applications of low to moderate complexity. Obtaining the transducer from a Petri net depends on obtaining its reachability graph as detailed in [Chapter 6](#), whose computational cost increases steeply with the size of the Petri net. A further limitation is that even for sufficiently small specifications, the

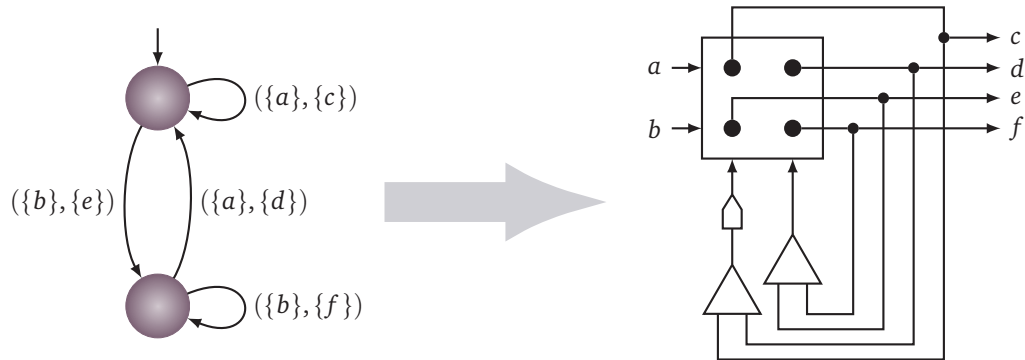


Figure 15.1: A transducer with two states and two inputs is synthesized using a 2-by-2 decision wait whose left column is associated with the initial state.

algorithm to be proposed can be outdone by a skilled human designer in some cases. For example, a behaviorally equivalent dual rail buffer cell to the one in [Appendix F](#) could be synthesized easily enough automatically, but at a higher cost. This limitation is a consequence of the algorithm being mostly agnostic about the set of primitives, hence unable to make strategic use of a SHUNT among other things. A third limitation, albeit a minor one, is that the algorithm sometimes opts for a refinement to the specification rather than an exact behavioral equivalent under certain unusual circumstances to be noted in [Section 15.2.2](#). A refinement nevertheless is acceptable in an implementation because it always meets the specification as explained in [Chapter 4](#). In summary, this method is best viewed as a tool for small non-critical parts of a larger project that affords some protection against putting human circuit designers out of a job.

The rest of this chapter consists of an intuitive overview of state based synthesis in [Section 15.1](#), and a derivation of a transducer representation amenable to synthesis in [Section 15.2](#) followed by a basic but often suboptimal algorithm in [Section 15.3](#). Readers in a hurry can skip [Section 15.2.2](#) if they are concerned only with deterministic processes, and might take away something actionable even by skipping everything after [Section 15.3](#). An optimizing input stage is discussed in [Section 15.4](#), and an optimization relevant to single-state transducers follows in [Section 15.5](#). The chapter concludes in [Section 15.6](#) with an optimization based on decomposing the specification into concurrent components where possible.

15.1 Overview

Getting state based synthesis exactly right is no mean feat, so it is best dispel any illusions about the subtleties involved at the outset before even thinking about formal algorithms.

15.1.1 The uncomplicated case

The core idea of state based synthesis is illustrated in [Figure 15.1](#). A planar decision wait has one row for each input and one column for each state of a transducer representing the process specification to be implemented. The column associated with the initial state starts with its column input enabled by a PUSH. When an external input arrives, the decision wait output specific to that

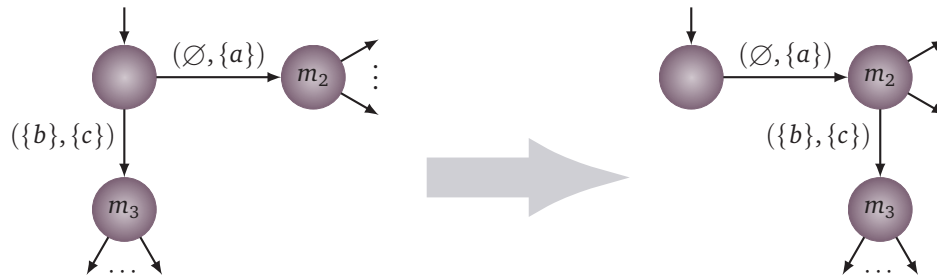


Figure 15.2: a process that is optionally non-quiescent, left, and an anti-refinement, right, showing an input-guarded edge and its terminus shifted to the terminus of the other edge

input and the current state drives a FORK whose outputs transmit the appropriate externally visible acknowledgment and enable the next state, perhaps by way of a MERGE. For example, an input of b when the system is in the initial state triggers the lower left output on the decision wait, which causes a FORK to output e and a MERGE to enable the alternative state. The edge labeled $(\{b\}, \{e\})$ from the initial state to the alternative in the transducer model indicates this behavior.

15.1.2 Complications

How is it possible to fill up a whole chapter with such a simple idea? Generalizing from this example to a realistic state based synthesis algorithm entails various technicalities.

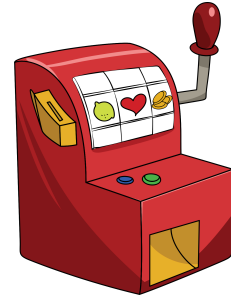
- For most transducers in practice, not every combination of states and inputs is valid. This issue is addressed for the most part by planar sparse decision waits (Section 11.4), but some states might not admit any valid inputs at all, which would imply a decision wait with an empty column unless there is some provision to the contrary, such as omitting the column for those states.
- Many specifications allow concurrent inputs, but a decision wait can cope with only one row input at a time. This issue can be resolved only by a front end stage performing the necessary arbitration (Section 13.4.1) and a transformation to a serialized transducer model along the lines of Section 7.3.
- Some specifications allow non-deterministic behavior. An implementation in this spirit would require not just a FORK and MERGE network as shown, but a full transcoder featuring a non-degenerate randomizing stage (Section 13.5.1).
- Furthermore, how do we know that a should connect to the first input and b to the second on the circuit generated in Figure 15.1 instead of the other way around, when their effects are not interchangeable? In practice, a synthesis algorithm requires not just a process specification but an alphabet ordering as input (Section 8.7.1).

15.1.3 Non-quiescent processes

More trouble comes from processes that are not initially quiescent. In the case of an initially non-quiescent process, the implementation must transmit one or more output signals initially before it receives any input signals, requiring a PUSH somewhere other than the one shown in Figure 15.1. Moreover, what are we to make of a process that decides at random whether to be quiescent, such as the one shown at the left of Figure 7.6, or worse yet, one whose choice is influenced by the environment, as in Figure 15.2 at the left? As a reminder, the exact protocol specified by the left process in Figure 15.2 guarantees that the process emits the signal a if the environment waits long enough, but the environment may also opt to transmit b instead of waiting, in which case the process may either acknowledge b with c and change to the state labeled m_3 , or may transmit a anyway because it has already decided to do so, and then respond to b in whatever way the state labeled m_2 prescribes.

Modified transducers

Deliberately specifying a process to make a non-deterministic choice about whether to output as in Figure 7.6 may be a bad idea, but ours is not to question why, and if that were the only issue, then synthesizing its implementation would pose no more of a problem than any other non-deterministic output. However, the issue isolated in Figure 15.2 needs further consideration. One way to approach it is by transforming the transducer on the left to the one on the right. By transplanting the input-guarded outgoing edges from every state to the termini of their unguarded sibling edges (and iterating if necessary), we obtain a transducer whose every adjacency set contains only one kind of edge or the other, so that every state is unequivocally quiescent or not. As implied above, only the non-deadlocked quiescent states would need explicit representation as columns of the decision wait, with equivalent output effects having been reassigned to their succeeding states by this transformation.



Refinement relationships

It is important to ask whether the transducer modified according to Figure 15.2 still meets the original specification. If the environment postpones transmitting an input until after it receives the initial output, then the subsequent behavior of the process is indistinguishable in either case. However, if the environment transmits an input before receiving an initial output, the effect on the process specified by the modified transducer is undefined, whereas the original has a required acknowledgment. A more formal explanation is that $\langle b, c \rangle$ is a quiescent trace of the original process, but a divergence of the modified process, whose divergences therefore also include all possible traces prefixed by $\langle b, c \rangle$. In terms of the theory explained in Section 7.4.3, the relational trace set of the latter is a superset of that of the former, so the original transducer expresses a refinement of the modified transducer.

A refinement relationship between a specification and its implementation normally would be acceptable in lieu of equivalence were it not for their roles being reversed, as they are in this case with the specification refining the implementation. Nevertheless, they are equivalent within a restricted environment that always withholds further input until after any enabled outputs are observable, and such an environment is implicit in the back end FORK and MERGE network in Figure 15.1. Because



Figure 15.3: a correct but suboptimal implementation (left) and an irredeemably wrong implementation (right) of the process in [Figure 7.8](#)

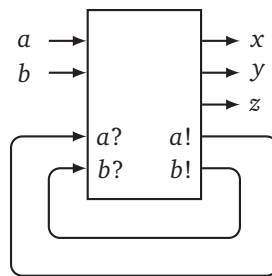


Figure 15.4: vague but potentially correct and optimal implementation of the process in [Figure 7.8](#)

an output signal from the decision wait delimits a change in state before reaching the column input enabling the next state, and because the row input signal is buffered in effect on the wire until then, the result is the same as if the environment had withheld the input signal until after the change from the initial state.

15.1.4 Non-deterministically concurrent processes

We are now ready for anything, or so it would seem until we recall the example of [Figure 7.8](#). The process depicted in the figure must acknowledge an input of a with an output of x and an input of b with an output of y , but may optionally acknowledge concurrent inputs of a and b with an output of z . As noted in [Section 7.3](#), the serial transducer derived from it by the algorithm previously proposed does not in itself faithfully model the original behavioral specification, so an implementation of the serial transducer in the style of [Figure 15.1](#) even with an arbitrating front end stage would be incorrect.

A more general method of circuit synthesis is needed to cover this case. One way of implementing this process would be by a circuit consisting of a wire from a to x , a wire from b to y and a floating or grounded output terminal for z as shown at the left of [Figure 15.3](#). This implementation meets the specification in a pedantic sense but never avails itself of the option to output z when the inputs are concurrent, making it most likely suboptimal in practice. A naive attempt to do better might lead to something like the implementation on the right of [Figure 15.3](#). This circuit outputs z when there are concurrent inputs, but also outputs x and y regardless. Worse yet, it diverges whenever it receives the same input twice unless there is both an intervening receipt of the other input and

current state	input	next state	output
1	a	2	$a!$
	b	3	$b!$
2	b	4	z
	$a?$	1	x
3	a	5	z
	$b?$	1	y
4	a	2	—
	b	6	$b!$
	$a?$	1	—
5	a	7	$a!$
	b	3	—
	$b?$	1	—
6	a	8	z
	$a?$	3	—
	$b?$	4	y
7	b	8	z
	$a?$	5	x
	$b?$	2	—
8	a	7	—
	b	6	—
	$a?$	5	—
	$b?$	4	—

Table 15.1: state table for the black box in Figure 15.4, with the initial state numbered 1

signals observed on all three outputs. Short of introducing a 5-terminal DI primitive (such as the RCEL [79]), no amount of tweaking can rescue a design starting down this road.¹

Figure 15.4 suggests a more promising approach. Here we envision a device with output terminals $a!$ and $b!$ wired to inputs labeled $a?$ and $b?$ respectively, so that none of $a!$, $b!$, $a?$ or $b?$ is exposed to the environment. The only externally visible terminals are labeled a , b , x , y and z . Starting from an initially quiescent state, the device outputs $a!$ if it receives an input on a and outputs $b!$ if it receives an input on b . (If it ever receives more than one input concurrently, it arbitrates between them.) If the next input after an output of $a!$ is $a?$, then it outputs x , and if the next input after an output of $b!$ is $b?$, then it outputs y . However, if it receives an input of b after emitting $a!$ or receives a after emitting $b!$, then it outputs z . By deliberately creating a race to be resolved by arbitration between $a?$ and b or between $b?$ and a , the circuit is able to distinguish between sequential and sufficiently concurrent external inputs of a and b with no deadlocks or spurious outputs.

¹or the author is a monkey's uncle

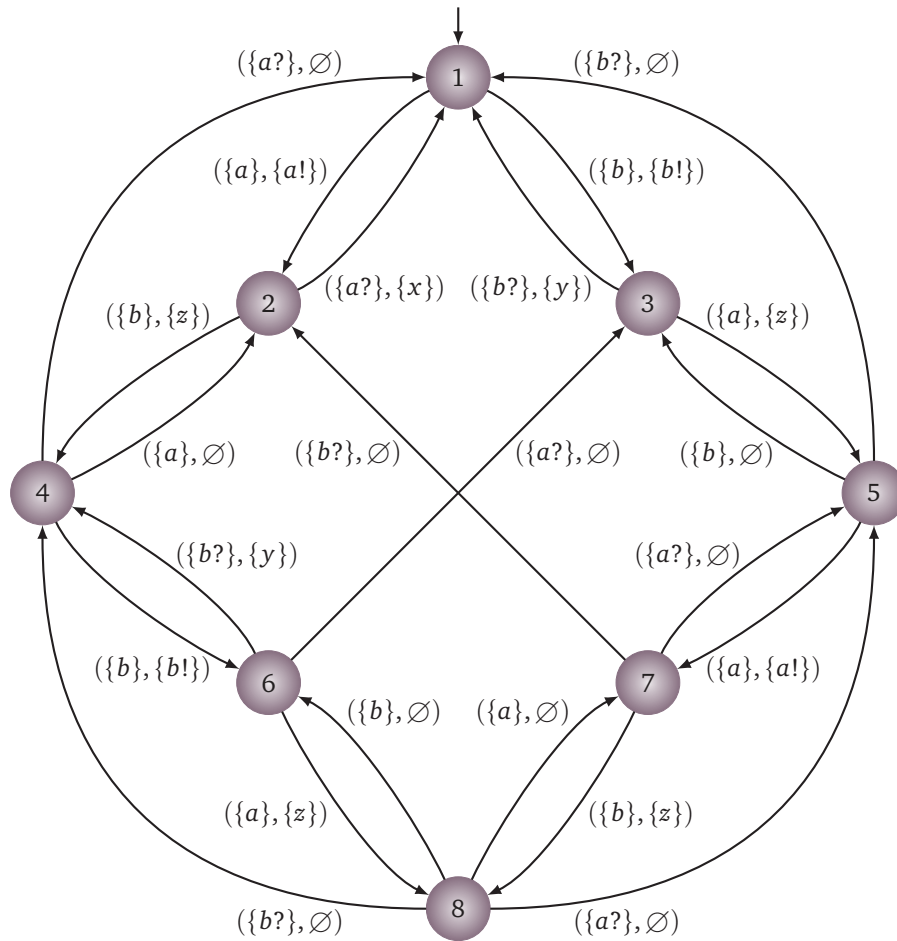


Figure 15.5: transducer whose state table is listed in Table 15.1

This behavior would be trivial to implement by a 4-way sequencer and a circuit comparable to the one in Figure 15.1 were it not for a few loose ends. Having transmitted z , the circuit is not fit to resume its initial state because it must anticipate the input $a?$ or $b?$ still in transit, and usually ignore it when it arrives. However, if there is a subsequent external input of a following an output of z and a previous $a!$ yet to arrive on $a?$, then the circuit must refrain from emitting another $a!$ so as not to interfere with the one in transit over the wire. Rather than ignoring the next $a?$ in this case, it can interpret it normally. Similar conditions apply to b , $b?$ and $b!$, and there are further edge cases to consider even beyond these. Nevertheless, they are not insurmountable if we enumerate every possible state carefully as in Table 15.1.

As Figure 15.5 shows, there is no impediment to basing a transducer model on Table 15.1 even if not every combination of states and inputs is valid, and even if there is not always an observable output associated with every state transition. The single-state transducer in Figure 7.8 has ballooned

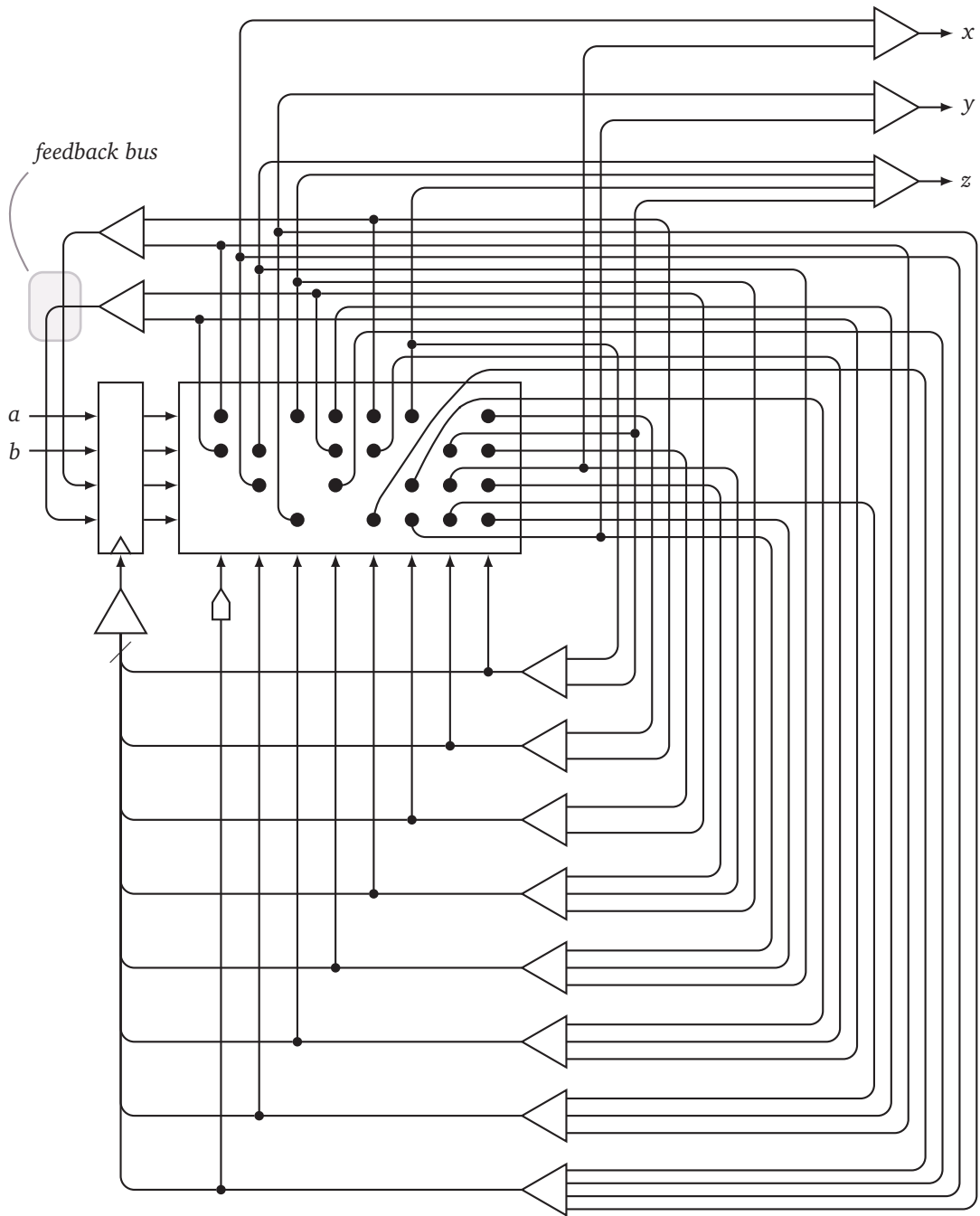


Figure 15.6: implementation of the process in [Figure 7.8](#) based on [Figure 15.4](#) and [Table 15.1](#), with the left column of the decision wait corresponding to state 1 and the right to state 8

into eight states, but the issue of non-deterministic concurrency is now resolved and a more specific description of the circuit in Figure 15.4 in the style of Figure 15.1 is obtained in Figure 15.6.

The complexity of the revised transducer and the synthesized circuit for what is nearly the simplest possible example of non-deterministic concurrency hints at the awkwardness of attempting this transformation by hand in general. Instead, we look to incorporate it as a phase of the circuit synthesis algorithm.

15.2 Transducer types

In a broad outline of state based synthesis, a given transducer model should be transformed first to the form described in Section 15.1.3 to cope with non-quiescence, which is called the *anti-refined transducer* for the duration of this chapter, then to a form comparable to Figure 15.6 if necessary to cope with non-deterministic concurrency, which is called the *feedback anti-refined transducer* hereafter, and then transformed at last as depicted in Figure 15.1 to the implementation. A derivation of the anti-refined transducer follows in Section 15.2.1, and a derivation of the feedback anti-refined transducer follows in Section 15.2.2.



15.2.1 Anti-refined transducers

To dispense first with a formal account of the transformation illustrated in Figure 15.2 needed as a prerequisite to state based synthesis, we denote the anti-refined transducer of a process $X \in \mathbb{D}$ by $\mathbf{AT} X$ to distinguish it from the usual transducer $\mathbf{T} X$ defined by Equation 7.12, in terms of a function

$$\mathbf{AT} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times (\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})$$

yet to be determined, provided the function yields a transducer with these properties.

- All states are quiescent except possibly the initial state.
- If the initial state is not quiescent, then it has no input-guarded outgoing edges.
- $\mathbf{AT} X$ is behaviorally equivalent to $\mathbf{T} X$ except possibly in its initially non-quiescent state, where it is allowed to diverge given any input.

The first point is already guaranteed by definition of the transducer according to Equation 7.8, which effectively precludes empty input bursts labeling any edges other than those originating from the initial state. To recapitulate the argument informally, each edge in the transducer $g = \mathbf{T} X$ labeled by an empty input burst can be transplanted to a predecessor of the edge's origin while remaining connected to the same terminus subject to a certain adjustment to the edge's input and output burst labels. In some cases the input burst labeling the edge might not be empty after the edge is transplanted, but if it still is, then it can be transplanted repeatedly until either its input burst is non-empty or it originates at the initial state.

More formally, let $(m, e) \in g$ denote the predecessor vertex of a state $n \in \mathcal{R}(e)$ connected to the state m by an edge $((i, o), n) \in e$. Suppose an outgoing edge $((j, k), l) \in (\Psi g) n$ from the state n is labeled by an empty input burst $j = \emptyset$. Then an edge $((i, o \cup k), l)$ would be created directly from

the predecessor m of n to the successor l of n with the new label $(i, o \cup k)$ derived from the labels (i, o) on the edge from m to n and (\emptyset, k) on the edge from n to l . A transducer

$$g' = \prod \bigcup_{(m,e) \in g} \{m\} \times (\rho \lambda((i, o), n). \bigcup_{((j,k),l) \in ((\Psi g) n) \cap ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})} \{(i, o \cup k), l\}) e$$

containing all edges possible to create in this way is then behaviorally equivalent to the transducer

$$(\rho \lambda(m, e). \prod \bigcup_{n \in \mathcal{R}(e)} \{n\} \times \bigcup_{t \in ((\Psi g') n) - ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})} \{t\}) \{(1, (\Psi g') 1)\}$$

pruned from it by deleting all edges labeled by empty input bursts other than those originating from the initial state. We can assume without loss of generality that any transducer $g = \mathbf{T}X$ is of this form.

It remains to effect the transformation illustrated in Figure 15.2. Part of this operation consists of adding every edge $t \in e$ labeled by a non-empty input burst in the adjacency set e of the initial state to the adjacency set $(\Psi g) n$ of every state $n \in \mathcal{R}(e)$ connected to the initial state by an edge labeled by an empty input burst. This change can be made by rewriting the transducer $g = \mathbf{T}X$ to $T_0 \mathbf{T}X$ with a function T_0 defined as follows.

$$T_0 = \lambda g. (\rho \lambda(m, e). \prod \bigcup_{((i,o),n) \in e} \{n\} \times ((\Psi g) n \cup \langle \emptyset, \bigcup_{t \in e - ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})} \{t\} \rangle_{\delta \emptyset})) \{(1, (\Psi g) 1)\}$$

The rest of this operation consists of deleting all edges labeled by non-empty input bursts from the adjacency set of the initial state if there are any edges labeled by empty input bursts. This effect is achievable by rewriting e to $e \cap s$ for

$$s = ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})$$

denoting the set of edges with empty input bursts whenever $e \cap s$ is non-empty, but leaving e unchanged otherwise. We therefore define the anti-refined transducer in general as

$$\mathbf{AT}(X) = (\lambda(I, O, N). \Phi \prod \bigcup_{(m,e) \in T_0 \mathbf{T}(I, O, X_{\mathbb{P}} N)} \{m\} \times (\lambda s. \langle e \cap s, e \rangle_{\delta e \cap s}) ((\{\emptyset\} \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})) X \quad (15.1)$$

with possible optimizations by Equation 6.12 and Equation 9.7 where applicable.²

15.2.2 Feedback anti-refined transducers

Mapping the single-state transducer in Figure 7.8 to the corresponding feedback anti-refined transducer in Figure 15.5 is just one instance of a problem tackled more generally in this section as a necessary prerequisite to state based synthesis of non-deterministically concurrent processes. The transformation to the feedback anti-refined transducer is reminiscent of the transformation to a serial transducer described in Section 7.3 in that it reduces every input burst to at most one member, but more complicated because it entails the creation not only of additional states but additional input and output symbols (e.g., the symbols $a?$, $a!$, $b?$ and $b!$ in Table 15.1). These newly created

²Only local Petri net optimizations by $\chi_{\mathbb{P}}$ are appropriate because converting X to canonical form by $\chi_{\mathbb{D}}$ (Equation 9.10) would probably waste more effort than it saves.

symbols are called **feedback symbols** for the sake of further discussion because they are created in pairs meant to be wired directly from output terminals to corresponding input terminals on the result as in [Figure 15.4](#) and [Figure 15.6](#). On the other hand, the transformation is easier insofar as only the input bursts and not the outputs need to be serialized. Because this transformation is assumed to be used only within a context of circuit synthesis and not verification, we also allow it to introduce refinements along the way when doing so is easier than precisely maintaining behavioral equivalence. These refinements tend to rule out deadlock when it is allowed but not required. For example, either of the transducers in [Figure 7.9](#) can map to the same feedback anti-refined transducer because it matters only for the resulting circuit to meet both specifications.

Input burst subsets

This transformation concerns transducers g wherein a vertex $(m, e) \in g$ might have an outgoing edge $((i, o), n) \in e$ whose input burst i is a proper subset of the input burst on some other member of the same adjacency set e . We can write $(T_1 g) m \in \mathcal{P}(\mathcal{P}(\mathbb{T}))$ for the set of all input bursts meeting this condition with respect to m with T_1 given by

$$T_1 = \lambda g. \lambda m. (\lambda e. \mathcal{D}(\mathcal{D}(e)) \cap \bigcup_{((i,o),n) \in e} \mathcal{P}(i) - \{i, \emptyset\}) (\Psi g) m. \quad (15.2)$$

Feedback symbols

Any input burst $i \in (T_1 g) m$ requires a dedicated pair of feedback input and output symbols to represent it in the alphabet of the feedback anti-refined transducer. To avoid clashes between these synthetic symbols and any already present in the alphabet, we reserve the generic terminals $\mathbb{G}^{\circ-1}(l)$ and $\mathbb{G}^{\circ-1}(l+1)$ for this purpose by [Equation 8.27](#) for an ordinal $l = (T_2(I, O) g) i$ given by

$$T_2 = \lambda(I, O). \lambda g. \lambda i. 2(1 + \max(\{0\} \cup (\mu \mathbb{G}^\circ) (\mathbb{G} \cap (I \cup O)))) + ((\mu^2 \eta) \mathcal{P}(I))^\circ (\mu \eta) i \quad (15.3)$$

where $I, O \in \mathcal{P}(\mathbb{T})$ are the input and output alphabets of the process, and $\eta : I \rightarrow \mathbb{N}$ is a total injective function of the input alphabet as explained in [Section 5.2.3](#).

Input bursts i outside of $(T_1 g) m$ require no feedback because their effect is fully determined immediately upon receipt. A convenient way to note this distinction is to identify a list \emptyset^2 of two empty sets with input bursts that do not need any feedback, and to identify a list

$$(\mu \mathbb{G}^{\circ-1})^* \langle \{l\}, \{l+1\} \rangle$$

of two unit sets of generic terminals having ordinals obtained as above with those that do. The result in either case for a state m and an input burst i of a transducer g with alphabets I and O is then expressible as

$$((T_3 \langle T_1, T_2(I, O) \rangle) g) (m, i) \in \mathcal{P}(\mathbb{G})^2$$

in terms of a function T_3 given by

$$T_3 = \lambda t. \lambda g. \lambda(m, i). (\mu \mathbb{G}^{\circ-1})^* (\lambda k. \langle \emptyset^2, (\lambda l. \langle \{l\}, \{l+1\} \rangle) (t_1 g) i \rangle_k) \delta_{\emptyset}^{\{i\} - (t_0 g) m}. \quad (15.4)$$

State transitions

Relaxation of the requirement for this transformation to preserve behavioral equivalence enables a succinct behavioral description of the transducer g in terms of a function

$$T_4 = \lambda g. \lambda m. \Psi \bigcup_{((i,o),n) \in (\Psi g) m} \{(i, \{(o, n)\})\} \quad (15.5)$$

whereby $((T_4 g) m) i \in \mathcal{P}(\mathcal{P}(\mathbb{T}) \times \mathbb{N})$ expresses the set of possible output bursts o and successor states n upon receipt of an input burst i from a current state m , and a function

$$T_5 = \lambda g. \lambda(m, i). (\bigcup (\mathcal{D}(\mathcal{D}((\Psi g) m)) - \mathcal{P}(\mathbb{T} - i))) - i \quad (15.6)$$

expressing by $(T_5 g) (m, i) \in \mathcal{P}(\mathbb{T})$ the set of acceptable input signals remaining when the transducer g starting from a state m has already received the set of inputs i . Normally the set of received inputs does not fully determine the set of acceptable inputs remaining because a transducer having received a subset of an input burst could commit to a path that prohibits the rest. A transducer that does the right thing regardless constitutes a refinement of the original specification, which we now allow as noted previously.

Intermediate representation

Moving on to the matter of building the graph of the feedback anti-refined transducer, we can benefit by representing it temporarily as a graph of states each encoded by a triple

$$(m, i, f) \in \mathbb{N} \times \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{G})$$

interpreted such that

- m is a state of the original transducer from which the feedback anti-refined transducer is to be derived
- i is a set of inputs acquired along a path from the state representing m to the state (m, i, f) in the feedback anti-refined transducer
- and f is a set of feedback inputs assumed to be currently in transit that should be effectively ignored by the feedback anti-refined transducer in state (m, i, f) .

For example, in state 4 shown in [Table 15.1](#), the feedback input a ? is ignored in the sense of having no observable effect beyond causing a change back to the initial state. For a finite alphabet, the set these triples is also finite, so a graph of them can be converted subsequently to a transducer with numerical states as usual. Postponing the details of the conversion for the moment, we stipulate only that states m of the original transducer be encoded as $(m, \emptyset, \emptyset)$ in the intermediate representation.

Adjacency sets

The benefit of this intermediate representation is that it clears a way to infer the whole adjacency set of any given state (m, i, f) , thereby making it easy to build the graph, subject only to the list of three functions

$$t = \langle t_0, t_1, t_2 \rangle = \langle T_3 \langle T_1, T_2(I, O) \rangle g, T_4 g, T_5 g \rangle$$

given by [Equation 15.2](#) through [Equation 15.6](#). As a reminder, for any state m and input burst i ,

- $t_0(m, i)_0$ contains the feedback input and $t_0(m, i)_1$ the output associated with i , if any
- $(t_1 m) i$ contains the pairs (o, n) of enabled output bursts and successor states
- and $t_2(m, i)$ contains the remaining acceptable inputs after i .

Edges in the adjacency set of a state (m, i, f) fit into five subsets $(\vec{e}_k t) (m, i, f)$ for k ranging from 0 through 4.

- Any acceptable input $j \in t_2(m, i)$ might be the last one needed to complete an input burst $i \cup \{j\}$ warranting a feedback output $l \in t_0(m, i \cup \{j\})_1$ and a change to a new state. The new state is one step further removed from m due to having received inputs $i \cup \{j\}$ and being prepared to ignore the feedback inputs $f \cup t_0(m, i)_0$. The possibility of a transition to this state is indicated by an edge in $(\vec{e}_0 t) (m, i, f)$ based on

$$\vec{e}_0 = \lambda t. \lambda(m, i, f). \bigcup_{j \in t_2(m, i)} \bigcup_{l \in t_0(m, i \cup \{j\})_1} \{((\{j\}, \{l\}), (m, i \cup \{j\}, f \cup t_0(m, i)_0))\}.$$

- If the transducer is already in a state (m, i, f) of having received an input burst i associated with a feedback input $l \in t_0(m, i)_0$, then the receipt of l indicates an absence of concurrent input, as in the example of $a?$ in state 2 of Table 15.1. In this case, the transducer should emit an ordinary output burst o and change to the corresponding successor state (n, \emptyset, f) for some enabled $(o, n) \in (t_1 m) i$. Edges expressing this behavior are members of $(\vec{e}_1 t) (m, i, f)$ for

$$\vec{e}_1 = \lambda t. \lambda(m, i, f). \bigcup_{l \in t_0(m, i)_0} \bigcup_{(o, n) \in (t_1 m) i} \{((\{l\}, o), (n, \emptyset, f))\}.$$

- An acceptable input $j \in t_2(m, i)$ for which the complete input burst $i \cup \{j\}$ warrants no feedback output because $t_0(m, i \cup \{j\})$ is empty enables a transition to a state $(n, \emptyset, f \cup t_0(m, i)_0)$ for some $(o, n) \in (t_1 m) (i \cup \{j\})$ that additionally ignores any feedback input in $t_0(m, i)_0$ associated with the current state. The transition from state 2 to state 4 in Table 15.1 is an example of this case, which is covered in general by members of $(\vec{e}_2 t) (m, i, f)$ as given by

$$\vec{e}_2 = \lambda t. \lambda(m, i, f). \bigcup_{j \in t_2(m, i)} (\lambda k. \langle \emptyset, \bigcup_{(o, n) \in ((t_1 m) (i \cup \{j\}))} \{((\{j\}, o), (n, \emptyset, f \cup t_0(m, i)_0))\} \rangle_k) \delta_{\emptyset}^{t_0(m, i \cup \{j\})_1}.$$

- An input $j \in t_2(m, i)$ that is acceptable but not sufficient to complete any input burst $i \cup \{j\}$ enabling a successor in $(t_1 m) (i \cup \{j\})$ nevertheless enables a change to an intermediate state one step further removed from m with no accompanying output burst. The intermediate state indicates having received $i \cup \{j\}$ and ignores any pending feedback inputs in $t_0(m, i)_0$ as well as those in f according to an edge belonging to $(\vec{e}_3 t) (m, i, f)$ for

$$\vec{e}_3 = \lambda t. \lambda(m, i, f). \bigcup_{j \in t_2(m, i)} (\lambda k. \langle \emptyset, \{((\{j\}, \emptyset), (m, i \cup \{j\}, f \cup t_0(m, i)_0))\} \rangle_k) \delta_{\emptyset}^{(t_1 m) (i \cup \{j\})}.$$

- Finally we allow a transition from any state having a non-empty set f of ignorable feedback inputs to a similar state having one less based on members of $(\vec{e}_4 t) (m, i, f)$ with

$$\vec{e}_4 = \lambda t. \lambda(m, i, f). \bigcup_{l \in f} \{((\{l\}, \emptyset), (m, i, f - \{l\}))\}.$$

To make use of these definitions for a transducer g , a list $t = \langle T_3 \langle T_1, T_2(I, O) \rangle, T_4, T_5 \rangle$ of three higher order functions, and a state $l = (m, i, f)$ of the feedback anti-refined transducer in its intermediate representation, we might abbreviate a unit set of vertices $\{(l, e)\}$ as

$$h(l) = \left\{ \left(l, \bigcup_{k=0}^4 \tilde{e}_k(t \triangle g^{\exists}) l \right) \right\}$$

by [Equation 12.3](#) in terms of a function temporarily denoted h to provide for an expression

$$(\rho \lambda(m, e). \bigcup_{(b,n) \in e} h n) h(1, \emptyset, \emptyset)$$

of the whole transducer obtained as a percolation by [Equation 6.4](#) from the initial state $(1, \emptyset, \emptyset)$, but perhaps better to summarize as $T_6 \langle T_3 \langle T_1, T_2(I, O) \rangle, T_4, T_5 \rangle g$ in terms of a function

$$T_6 = \lambda t. \lambda g. (\lambda h. (\rho \lambda(m, e). \bigcup_{(b,n) \in e} h n) h(1, \emptyset, \emptyset)) \lambda l. \left\{ \left(l, \bigcup_{k=0}^4 \tilde{e}_k(t \triangle g^{\exists}) l \right) \right\}. \quad (15.7)$$

State numbering

Converting the states (m, i, f) in the intermediate representation to natural numbers in the preferred representation is mostly a matter of mapping each triple to its ordinal with respect to this state space. This effect is partly achieved by constructing the set of pairs

$$((\mu \eta) i, f) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{G})$$

induced by all subsets of input bursts $i \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$ and all sets of feedback inputs

$$f \in \bigcup_{m \in \mathcal{D}(g)} \bigcup_{i \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))} ((T_3 \langle T_1, T_2(I, O) \rangle g) (m, i))_0$$

inferred from a transducer g with alphabets I and O by [Equation 15.2](#) through [Equation 15.4](#), which we can abbreviate as $(T_7 T_3 \langle T_1, T_2(I, O) \rangle) g$ in terms of a function

$$T_7 = \lambda t. \lambda g. \left(\bigcup_{i \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))} (\mu^2 \eta) \mathcal{P}(i) \right) \times \mathcal{P} \left(\bigcup_{m \in \mathcal{D}(g)} \bigcup_{i \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))} ((t g) (m, i))_0 \right). \quad (15.8)$$

In this way, the combination of a pair (i, f) with a function $t = T_7 T_3 \langle T_1, T_2(I, O) \rangle$ determines a unique natural number

$$(t g)^\circ ((\mu \eta) i, f) \in \mathbb{N}$$

and the combination of t with a triple (m, i, f) determines the unique natural number

$$m |t g| + (t g)^\circ ((\mu \eta) i, f)$$

provided m is a state, i is a subset of an input burst, and f is a set of feedback inputs for the transducer g , based on the total ordering on \mathbb{G} stipulated by [Equation 8.28](#). Offsetting this value to

$$1 + \max(\{0\} \cup \mathcal{D}(g)) + m |t g| + (t g)^\circ ((\mu \eta) i, f)$$

prevents it from clashing with any extant state $m \in \mathcal{D}(g)$. Assigning this number to (m, i, f) is therefore satisfactory in all cases other than i and f both empty, for which Equation 15.7 constrains the value to the original state m . Covering the latter case as well requires an assignment of

$$((T_8 T_7 T_3 \langle T_1, T_2(I, O) \rangle) g) (m, i, f) \in \mathbb{N}$$

to the state (m, i, f) based on a function

$$T_8 = \lambda t. \lambda g. \lambda (m, i, f). \langle 1 + \max(\{0\} \cup \mathcal{D}(g)) + m |t g| + (t g)^\circ ((\mu \eta) i, f), m \rangle_{\delta^{i \cup f}}. \quad (15.9)$$

Summary

Economizing somewhat on an expression of the feedback anti-refined transducer with renumbered states, let t temporarily denote the list of five functions

$$t = \langle T_3 \langle T_1, T_2(I, O) \rangle, T_4, T_5, T_6, T_8 \circ T_7 \rangle$$

based on Equation 15.2 through Equation 15.9 so that $(t_4 t_0) g : \mathbb{N} \times \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{G}) \rightarrow \mathbb{N}$ is the state renumbering function for a transducer g with alphabets I and O , and

$$t_3 \langle t_0, t_1, t_2 \rangle g \in \mathcal{P}((\mathbb{N} \times \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{G})) \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times (\mathbb{N} \times \mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{G}))))$$

is the feedback anti-refined transducer in its intermediate representation as a graph of triples. Then the anti-refined transducer with numerical states is expressible as $(T_9 t) g$ for T_9 defined by

$$T_9 = \lambda t. \lambda g. \Phi \bigcup_{(l,e) \in t_3 \langle t_0, t_1, t_2 \rangle g} \{((t_4 t_0) g) l, \bigcup_{(b,n) \in e} \{(b, ((t_4 t_0) g) n)\}\}$$

with state numbers reduced to more moderate consecutive values and any other optimizations possible by Equation 6.12. An expression for the feedback anti-refined transducer in terms of a function

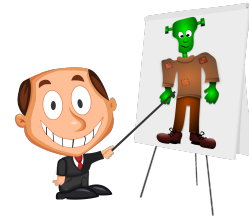
$$\mathbf{FAT} : \mathbb{D} \rightarrow \mathcal{P}(\mathbb{N} \times \mathcal{P}((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N}))$$

parameterized by process X and its anti-refined transducer by Equation 15.1 follows.

$$\mathbf{FAT}(X) = (\lambda(I, O, N). T_9 \langle T_3 \langle T_1, T_2(I, O) \rangle, T_4, T_5, T_6, T_8 \circ T_7 \rangle \mathbf{AT} X) X$$

15.3 Basic synthesis

The simplest practical generalization of the example in Figure 15.1 to unrestricted process specifications consists mainly of the three stages shown in Figure 15.7.



- The front end serializer stage corresponds to the sequencer in Figure 15.6, and is responsible for interfacing between the input bus from the environment shown at the left, which may transmit concurrent signals, and a 1-hot channel to the middle stage. Each concurrent input requires an acknowledgment from the back end controller stage before the serializer allows the next one through.

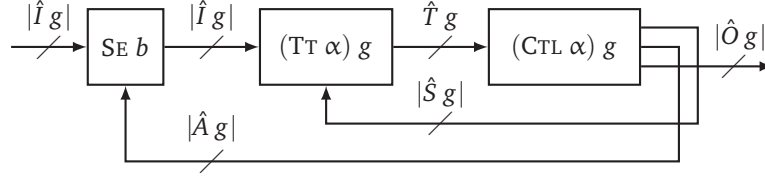


Figure 15.7: A basic state-based synthetic circuit $(\Omega_m^\alpha \cup_m^\alpha) g$ implementing a feedback anti-refined transducer g consists of a serializer, a transition table, and a controller block, where $\alpha \in \mathbb{T}^*$ is an alphabet ordering and $b \in \mathcal{P}(\mathbb{N})^*$ is a list of sets of sequencer input indices.

- The middle transition table stage corresponds to the decision waits in [Figure 15.1](#) and [Figure 15.6](#), with at most one column for each state of the feedback anti-refined transducer g , and a PUSH to the column of the initial state if the process specification is initially quiescent.
- The back end controller stage is responsible for transmitting the outputs to the environment associated with each state change, selecting the next state on the transition table stage, and acknowledging the input to the serializer stage if it is one of several concurrent inputs. The controller consists mainly of a transducer taking a 1-hot bus from the decision wait as input, but may also have a PUSH on an additional line if the process specification is not initially quiescent.

Not shown in [Figure 15.7](#) is the feedback bus indicated in [Figure 15.4](#) and [Figure 15.6](#). For processes whose feedback anti-refined transducer $g = \mathbf{FAT} X$ entails no alphabet symbols other than those of the anti-refined transducer $\mathbf{AT} X$, the feedback bus is absent and the whole implementation reduces to [Figure 15.7](#). In any case, we can postpone the discussion of the feedback bus and related details while focusing momentarily on these basic building blocks.

In addition to a process $X = (I, O, N) \in \mathbb{D}$ determining the transducer $g = \mathbf{FAT} X$ to be implemented, the synthesis algorithm is parameterized by an alphabet ordering $\alpha \in \mathbb{T}^*$ satisfying

$$\forall a \in I \cup O. |\alpha \uparrow \{a\}| = 1$$

as usual, meaning that every member of the input and output alphabets appears exactly once in α . The alphabet ordering is needed to associate each terminal on the circuit with a corresponding alphabet symbol. The first input symbol in α corresponds to the first input terminal on the circuit, the second symbol to the second terminal, and so on. The output terminals are ordered similarly.

The derivation of a transformation from a process $X \in \mathbb{D}$ to a circuit proceeds in part from a decomposition function

$$\mathcal{U}_m^\alpha : \mathcal{P}(\mathbb{N} \times ((\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T})) \times \mathbb{N})) \rightarrow \mathcal{P}(\mathbb{N})^*$$

induced by the alphabet ordering α and taking the transducer $g = \mathbf{FAT} X$ to a list

$$b = \mathcal{U}_m^\alpha g \in \mathcal{P}(\mathbb{N})^*$$

to keep a record of the possible ways the specification allows signals to arrive concurrently. Each term of b corresponds to a sequencer in an array implementing the serializer as shown in [Figure 15.8](#),

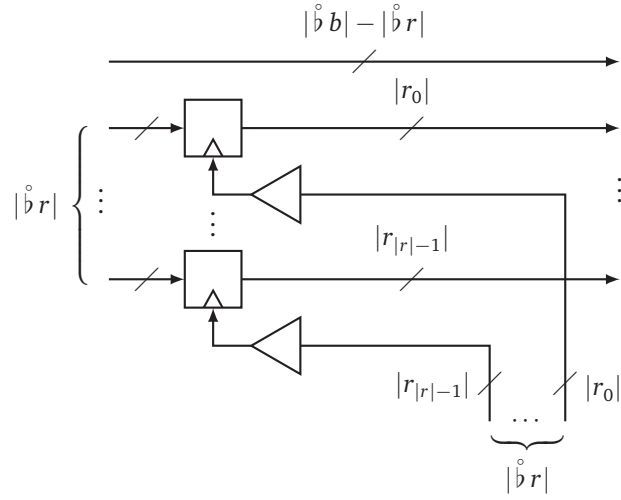


Figure 15.8: The front end serializer SE b in Figure 15.7 is made from sequencers and a bus, where r is given by Equation 15.13.

such that the ordinals with respect to α of any two inputs that can arrive concurrently belong to the same term.

To give a complete account of the basic circuit synthesis algorithm, Section 15.3.1 defines the decomposition function explicitly, Section 15.3.2 defines the three main building blocks shown in Figure 15.7, and Section 15.3.3 incorporates them into a combining form Ω_m^α enabling an expression of the result as $(\Omega_m^\alpha \cup_m^\alpha) \text{FAT } X \in \mathbb{H}$ for most valid process specifications X . To conclude, certain ill-conceived specifications featuring unsafe or unused inputs and outputs are addressed and the feedback bus is incorporated if necessary in Section 15.3.4.

15.3.1 Decomposition

To continue on the subject of unsafe inputs momentarily, a set $\hat{I}g \in \mathcal{P}(\mathbb{T})$ would be more indicative than the input alphabet I of the process $X = (I, O, N)$ of the required bus widths in the circuit if it were to contain only the input symbols that actually label an edge somewhere in the transducer $g = \text{FAT } X$. It is given by

$$\hat{I} = \lambda g. \bigcup \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g))) \tag{15.10}$$

and can differ from I not only due to the presence of feedback inputs, but due to inputs that are always unsafe. Normally the latter would be unintended, so a formal distinction between I and $\hat{I}g$ is worth preserving conspicuously. The set $\hat{O}g \in \mathcal{P}(\mathbb{T})$ is analogous,

$$\hat{O} = \lambda g. \bigcup \mathcal{R}(\mathcal{D}(\bigcup \mathcal{R}(g))) \tag{15.11}$$



where members of $O - \hat{O}g$ would be the outputs that are never used because the designer forgot about them.

More memorable perhaps is the current campaign to derive a decomposition $b = \mathcal{U}_m^\alpha g$ specifying the concurrency of the feedback anti-refined transducer g as mentioned above. The complementary problem to transforming a specification into a transducer with no more than one member in each input burst is that of embedding the result in an environment that serializes any possible concurrent inputs to it. One way of preventing any signals from reaching it concurrently would be to interpose a sequencer as a front end between the process and its original environment. According to the definition in [Section 13.4.1](#), the sequencer requires each signal that passes through to be acknowledged individually, so the process need never receive more than one at the same time.

This approach would be effective but may be more costly than necessary. If the process specification does not mandate any particular response to concurrent inputs, then there is no reason to guard against them. Its behavior is undefined when inputs are concurrent, so nothing an implementation might do in that event can violate the specification however surprising. On the other hand, if a process has a large number of inputs with concurrency prohibited among most of them but allowed on a small subset, then it would be correct and less costly to route only that small subset of inputs through a small sequencer and let the rest interface directly with the external environment. The ideal solution is the least costly one that meets any specification along this spectrum. Even if every input is concurrent with another, a solution using multiple small sequencers is preferable to a solution using a single large one if it exists.

Obtaining the best solution requires some knowledge of the combinations of concurrent inputs the process specification allows. Knowing that two signals can arrive concurrently enables us to route both through a common sequencer as shown in [Figure 15.8](#). Moreover, even if a given pair of signals can never arrive concurrently, they are inevitably bound to pass through the same sequencer if there is a third signal with which either of them can be concurrent at different times. In more mathematical terms, we seek a partition on the input alphabet induced not by a relation of mutual concurrency between pairs of inputs but by the transitive closure of this relation.

Although no input bursts in the feedback anti-refined transducer $g = \text{FAT } X$ contain multiple signals, we can infer that an input i is allowed according to the original specification $\text{AT } X$ to appear concurrently with another input j if there is an edge $((\{i\}, \emptyset), n)$ in $\bigcup \mathcal{R}(g)$ terminating at some state n wherein j is enabled, and crucially the edge is labeled by an empty output burst. The concurrent inputs would be given in general by the function

$$\lambda i. \bigcup_{(b,n) \in (\bigcup \mathcal{R}(g)) \cap (\{\{i\} \times \{\emptyset\}\} \times \mathbb{N})} \bigcup_{((j,k),l) \in (\Psi g) n} \{j\}$$

of an arbitrary input $i \in \hat{I} g$. All members of an input burst $b \in \mathcal{P}(\hat{I} g)$ are concurrent at a minimum with any members of the set

$$(\rho \lambda i. \bigcup_{n \in \mathcal{R}((\bigcup \mathcal{R}(g)) \cap (\{\{i\} \times \{\emptyset\}\} \times \mathbb{N}))} \bigcup_{((j,k),l) \in (\Psi g) n} \{j\}) b$$

obtained by traversing all succeeding edges labeled by empty output bursts, and all sets of sets of mutually concurrent inputs are obtained by mapping a related function over all input bursts $b \in \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$.

$$(\mu \rho \lambda i. \bigcup_{n \in \mathcal{R}((\bigcup \mathcal{R}(g)) \cap (\{\{i\} \times \{\emptyset\}\} \times \mathbb{N}))} \bigcup_{((j,k),l) \in (\Psi g) n} \{j\}) \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$$



In anticipation of synthesizing the circuit, it is more convenient to refer to the numerical ordinals of the inputs relative to the alphabet ordering α than their symbols, which are obtained by

$$(\mu^2 (\alpha \uparrow \hat{I} g)^{-1}) (\mu \rho \lambda i. \bigcup \bigcup \{j\}) \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$$

$$n \in \mathcal{R}((\bigcup \mathcal{R}(g)) \cap ((\{i\} \times \{\emptyset\}) \times \mathbb{N})) \quad ((j,k),l) \in (\Psi g) n$$

and to simplify this collection of sets to a partition by eliminating redundant subsets.

$$(\lambda t. t - \bigcup_{s \in t} \mathcal{P}(s) - \{s\}) (\mu^2 (\alpha \uparrow \hat{I} g)^{-1}) (\mu \rho \lambda i. \bigcup \bigcup \{j\}) \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$$

$$n \in \mathcal{R}((\bigcup \mathcal{R}(g)) \cap ((\{i\} \times \{\emptyset\}) \times \mathbb{N})) \quad ((j,k),l) \in (\Psi g) n$$

Finally, to transform this partition to a lexicographically ordered list $\mathcal{U}_m^\alpha g \in \mathcal{P}(\mathbb{N})^*$ of equivalence classes of input ordinals, we have the decomposition function

$$\mathcal{U}_m^\alpha = \lambda g. ((\lambda t. t - \bigcup_{s \in t} \mathcal{P}(s) - \{s\}) (\mu^2 (\alpha \uparrow \hat{I} g)^{-1}) (\mu \rho \lambda i. \bigcup \bigcup \{j\}) \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g))))^{\circ^{-1}}.$$

$$n \in \mathcal{R}((\bigcup \mathcal{R}(g)) \cap ((\{i\} \times \{\emptyset\}) \times \mathbb{N})) \quad ((j,k),l) \in (\Psi g) n \quad (15.12)$$

15.3.2 Building blocks

Working toward the construction of a combining form Ω_m^α capturing the basic synthesis method illustrated in Figure 15.7, we tackle the serializer, transition table, and controller block definitions in this section.

Serializer

The serializer consists of an array of sequencers with a multi-way MERGE connected to the acknowledgment input on each, and an individual bus line for each input that is never concurrent with any other. The sequencers and the bus are possible to express concisely in terms of a list $b = \mathcal{U}_m^\alpha g$ given by the decomposition derived above. Each term $h \in \mathcal{R}(b)$ with $|h| > 1$ calls for a sequencer $\text{SEQ } |h|$, but each singleton set h indicates only a wire $\mathbb{1}$. Hence the whole array given by the fold

$$(\mathcal{F}_{Z1} \lambda(h, t). \langle \mathbf{R}(\text{SEQ } |h| \downarrow \mathbb{1}, t) \uparrow \mathbb{1}, \mathbf{R}(\mathbb{1}, t) \rangle_{\delta_1^{|h|}} b$$

with inputs rolled as shown aggregates the sequencer request inputs ahead of the acknowledgments following in reverse order. A permutation

$$p = (\overset{\circ}{b} b)^{-1} \in \mathbb{N}^*$$

specifying matching front and back permutation networks on a block

$$p \times (\mathcal{F}_{Z1} \lambda(h, t). \langle \mathbf{R}(\text{SEQ } |h| \downarrow \mathbb{1}, t) \uparrow \mathbb{1}, \mathbf{R}(\mathbb{1}, t) \rangle_{\delta_1^{|h|}} b \times p$$

arranges the requests and acknowledgments according to the alphabet ordering α regardless of their particular concurrency relationships. We still need to construct the MERGE network for the



acknowledgments, but at least the requests and grants are covered by a block denoted $\text{SRG } b \in \mathbb{H}$ in terms of a function $\text{SRG} : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ given by

$$\text{SRG} = \lambda b. (\lambda p. p \times (\mathcal{F}_{\mathbf{Z1}} \lambda(h, t). \langle \mathbf{R}(\text{SEQ } |h| \downarrow \mathbf{1}, t) \uparrow \mathbf{1}, \mathbf{R}(\mathbf{1}, t) \rangle_{\delta_1^{|h|}}) b \times p) (\overset{\circ}{b} r)^{-1}.$$

The acknowledgment signals feeding back from the controller in [Figure 15.7](#) to the serializer funnel through a MERGE network featuring one multi-way MERGE for each sequencer. As noted above, not every term in $h \in \mathcal{R}(b)$ corresponds to a sequencer, only the terms $h \in \mathcal{R}(r)$ with

$$r = b (\lambda s. \langle s \rangle \ll \delta_1^{|s|})^* b \in \mathcal{P}(\mathbb{N})^* \quad (15.13)$$

being the sublist whose terms each contain more than one element. The MERGE network, which could be empty, follows as

$$(\mathcal{F}_{\mathbf{Z1}} \mathbf{R}) (\lambda h. \text{MERGE } |h|)^* r$$

but its inputs need reordering by a permutation network for consistency with the rest of the serializer, whose inputs are arranged according to their ordinals in b . A permutation derived from a flattening of r

$$(\mathcal{R}(\overset{\circ}{b} r)^{\circ})^* \overset{\circ}{b} r \in \mathbb{N}^*$$

renumbers $\overset{\circ}{b} r$ to obtain a bijection as required while otherwise preserving its order. A network that receives the serializer acknowledgments is expressible in terms of a function $\text{SA} : \mathcal{P}(\mathbb{N})^* \rightarrow \mathbb{H}$ defined by

$$\text{SA} = \lambda r. ((\mathcal{R}(\overset{\circ}{b} r)^{\circ})^* \overset{\circ}{b} r)^{-1} \times (\mathcal{F}_{\mathbf{Z1}} \mathbf{R}) (\lambda h. \text{MERGE } |h|)^* r$$

as $\text{SA } r \in \mathbb{H}$ on behalf of the serializer $\text{SE } b \in \mathbb{H}$ defined in terms of a function

$$\text{SE} = \lambda b. (\lambda r. (\mathbf{Z}^{|r|} \mathbf{R}(\text{SA } r, \text{SRG } b)) \uparrow | \overset{\circ}{b} r |) b (\lambda s. \langle s \rangle \ll \delta_1^{|s|})^* b. \quad (15.14)$$

Each MERGE output from $\text{SA } r$ connects to the corresponding sequencer acknowledgment input in $\text{SRG } b$, and the acknowledgment inputs are rolled to a position following the requests.

Transition table

The next block in the construction, the transition table, is mostly a planar sparse decision wait possibly with a PUSH attached, whose coordinates encode the input feedback anti-refined transducer model of the process being implemented.

A good start is to observe from [Figure 15.1](#) and the related discussion that each edge in the transducer graph corresponds to a unique output terminal on the decision wait. The columns correspond roughly to the states in the transducer and the rows to the inputs. Any given column exhibits an output terminal for each input-labeled outgoing edge from the corresponding state, whose row coordinate is determined by the input labeling the edge. Hence the number of rows in the decision wait is precisely $|\hat{I} g|$.

Similar precision as to the number of columns would be useful. Following the transformation illustrated in [Figure 15.2](#), any non-quiescent state is left with no input-labeled outgoing edges, so if there were exactly one column for each state $m \in \mathcal{D}(g)$, the column for each non-quiescent state would exhibit no output terminals. The same would be true of any deadlocked state. However, decision waits of this form are disallowed in [Section 11.1.4](#). As suggested previously, we have to insist on restricting the columns to non-deadlocked quiescent states $m \in \hat{S} g$ by

$$\hat{S} = \lambda g. \mathcal{D}(\{(m, e) \in g \mid \bigcup \mathcal{D}(\mathcal{D}(e)) \neq \emptyset\}) \quad (15.15)$$

and abide by whatever implications follow.

Normally when the circuit executes a change of state, the controller block yet to be specified is expected to transmit a signal to the decision wait enabling the column representing the succeeding state, but this action is impossible if the succeeding state $m \in \mathcal{D}(g) - \hat{S}g$ corresponds to no column in the decision wait. Designing the controller with one less output poses no problem, but where does that leave the rest of circuit? There are three cases to consider.



1. For a deadlocked succeeding state (one with no outgoing edges at all), the decision wait would block any further progress because no column would be enabled, even if another signal were to arrive on a row input. Although this behavior is perhaps undesirable in itself, it conforms nicely to that of a deadlocked transducer, so all is well.³
2. A non-quiescent successor would be problematic, but it is reasonable to ignore this possibility because a transducer constructed according to [Section 7.2.2](#) and [Section 15.2.1](#) can not have a non-quiescent state except for the initial state. Any incident edge on a non-quiescent state would be rewritten to bypass it, making it unreachable and subject to pruning unless it is the initial state.
3. For a non-quiescent initial state, this same lack of predecessors means it is never reached again through any state transition during subsequent operation. In this case, the controller can be designed to begin by actively enabling a successor to the initial state and emitting the outputs associated with an initial state transition.

The conclusion is that we can implement the transition table with one column for each member of $\hat{S}g$ as if those were the only states, and that even the initialization by a PUSH can be omitted if the initial state is not among them.

Decision wait Recalling that a planar sparse decision wait is specified by a set $c \in \mathcal{P}(\mathbb{N}^2)$ of points each having a row and a column coordinate, we have for each state $m \in \hat{S}g$ and each input symbol $r \in \bigcup \mathcal{D}(\mathcal{D}(e))$ indicated by the adjacency set e in $(m, e) \in g$ the list of two coordinates

$$\langle (\alpha \uparrow \hat{I}g)^{-1} r, (\hat{S}g)^\circ m \rangle$$

where $(\alpha \uparrow \hat{I}g)^{-1} r$ gives the row number in terms of the input symbol r by its position within the alphabet ordering α , and $(\hat{S}g)^\circ m$ gives the column number in terms of the state m relative to the other quiescent non-deadlocked states in $\hat{S}g$. A sparse decision wait $s = \text{SDW } c \in \mathbb{H}$ is then expressible in terms of the set of points

$$c = \bigcup_{(m,e) \in g \cap ((\hat{S}g) \times \mathcal{R}(g))} (\mu \lambda r. \langle (\alpha \uparrow \hat{I}g)^{-1} r, (\hat{S}g)^\circ m \rangle) \cup \mathcal{D}(\mathcal{D}(e)) \quad (15.16)$$

by any of [Equation 11.12](#), [Equation 11.14](#), or [Equation 11.36](#) depending on how sophisticated one wants to be about sparse decision wait decomposition strategies. It would also suffice to write $(m, e) \in g$ in place of $(m, e) \in g \cap ((\hat{S}g) \times \mathcal{R}(g))$ in this expression because $\bigcup \mathcal{D}(\mathcal{D}(e))$ is empty when m is not a member of $\hat{S}g$.

³Technically it amounts to a refinement because the transducer in this state diverges upon further input whereas the circuit typically can be shown to deadlock at least initially, but an implementation that refines its specification is acceptable.

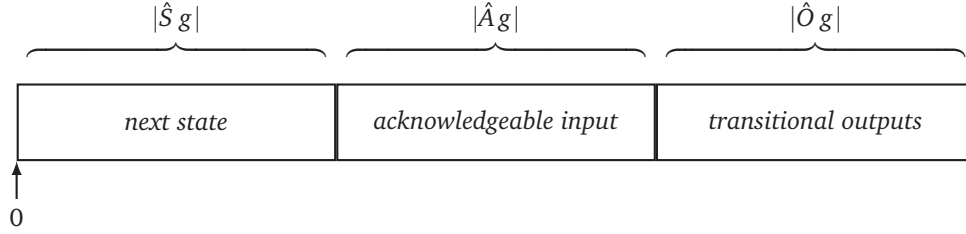


Figure 15.9: positions, widths, and meanings of the three fields in the controller's output bus format, with line number 0 at the left

Initialization In the usual case of an initially quiescent process, the decision wait needs a PUSH in series with the column input corresponding to the initial state to make up the transition table. Whenever the condition $1 \in \hat{S}g$ holds, meaning state number 1 is quiescent, that column would be numbered $(\hat{S}g)^\circ 1$, and the input terminal on the decision wait for that column would be numbered

$$i = |\hat{I}g| + (\hat{S}g)^\circ 1$$

to offset it by the number of rows $|\hat{I}g|$, for an overall transition table

$$\mathbf{F}\langle \text{PUSH}, s \uparrow i \rangle \downarrow i \in \mathbb{H}$$

in this case or

$$(\lambda k. \langle s, (\lambda i. \mathbf{F}\langle \text{PUSH}, s \uparrow i \rangle \downarrow i) (|\hat{I}g| + (\hat{S}g)^\circ 1) \rangle_k) \delta_{\emptyset}^{\{1\} - \hat{S}g} \in \mathbb{H}$$

to cover the unusual case $1 \notin \hat{S}g$ where no PUSH is needed. If we regard the conditional combination with a PUSH as a transformation TTI g from a decision wait s to an initialized transition table in terms of a function

$$\text{TTI} = \lambda g. \lambda s. (\lambda k. \langle s, (\lambda i. \mathbf{F}\langle \text{PUSH}, s \uparrow i \rangle \downarrow i) (|\hat{I}g| + (\hat{S}g)^\circ 1) \rangle_k) \delta_{\emptyset}^{\{1\} - \hat{S}g} \quad (15.17)$$

then an expression $(\text{TT} \alpha) g$ suffices for the transition table block in terms of the alphabet ordering α , the serialized transducer g , and a function defined as follows.

$$\text{TT} = \lambda \alpha. \lambda g. (\text{TTI} g) \text{SDW} \bigcup_{(m,e) \in g} (\mu \lambda r. \langle (\alpha \uparrow \hat{I}g)^{-1} r, (\hat{S}g)^\circ m \rangle) \cup \mathcal{D}(\mathcal{D}(e)) \quad (15.18)$$

Controller

The controller block outputs a word in three fields as shown in Figure 15.9, with one field for each of the transition table, the serializer, and the external environment. The first has a width of $|\hat{S}g|$ bus lines so that there is one line for each column of the decision wait in the transition table by Equation 15.15, and carries a 1-hot code. The next field consists of $|\hat{A}g|$ lines for \hat{A} defined by

$$\hat{A} = \lambda g. (\lambda p. \bigcup_{c \in p} \langle c, \emptyset \rangle_{\delta_1^{|c|}}) (\lambda t. t - \bigcup_{s \in t} \mathcal{P}(s) - \{s\}) (\mu \rho \lambda i. \bigcup \bigcup \{j\} \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g))) \quad (15.19)$$

where $\hat{A}g \in \mathcal{P}(\mathbb{T})$ can be viewed as the subset of “acknowledgeable” symbols in the input alphabet. This set is restricted to inputs that are allowed by the specification to arrive concurrently with other inputs, and are therefore made to pass through a sequencer in the serializer block, which then requires acknowledgment from the controller to enable further progress. Members of $\hat{A}g$ are inferred from the feedback anti-refined transducer g similarly to the decomposition defined by Equation 15.12, but are independent of any the alphabet ordering (cf. Equation 15.13). This field by itself is also a 1-hot code. The last field covers $|\hat{O}g|$ bus lines transmitting output signals either sequentially or concurrently to the external environment.

The controller is more complicated than the other blocks in this construction so its derivation is divided into several parts.

Transcoders A simple approach to the controller is to regard it primarily as an instance of a transcoder design problem and to rely on the solution given by Equation 13.22. It is not entirely a transcoder because it may need an additional PUSH if the specification to be implemented is not initially quiescent, but we may focus on the transcoding aspect of it for the moment.

Equation 13.22 provides an algorithm to generate a transcoder circuit from a set of pairs of input and output code words, so the current problem needs to be cast in those terms. Because the channel from the decision wait in the transition table to the controller transmits only one signal at a time, the input code is naturally a 1-hot code, with each input code word therefore a unit set. Because concurrent outputs may be needed, each output code word is a union of three sets, with one for each field illustrated in Figure 15.9. More specifically, each edge $((i, o), n) \in \bigcup \mathcal{R}(g)$ in the transducer graph determines a particular pair of input and output code words.

- The input word is the unit set containing the ordinal of that edge relative to the other edges according to the alphabet ordering, because this ordinal is the position of the output terminal on the decision wait in the transition table that issues the signal associated with the edge.
- The output word contains a number derived from the successor state n in the first field but only if n is a member of $\hat{S}g$, and a number derived from i in the second field offset by with width of the first field, but only if i is a non-empty subset of $\hat{A}g$, with either or both fields being empty otherwise. The last field contains all output ordinals derived from o offset by the widths of the first two fields.

Points As an intermediate step toward transforming the set of edges in a transducer to the set of pairs describing the transcoder circuit, we can transform it to a set s of points $p \in \mathbb{N}^2$ with

$$s = \bigcup_{(m,e) \in g} (\mu \lambda i. \langle (\alpha \uparrow \hat{I}g)^{-1} i, m \rangle) \cup \mathcal{D}(\mathcal{D}(e))$$

similar to the coordinate points in Equation 15.16 used to specify the decision wait in the transition table, but with absolute state numbers m retained. Each point $p = \langle p_0, p_1 \rangle \in s$ has a value of p_0 matching the ordinal of an input alphabet symbol labeling an edge originating from a state $m = p_1$. If the process specification is not initially quiescent, meaning the initial state 1 is not a member of $\hat{S}g$ because no inputs are enabled in it, then we should manually include $\langle |\hat{I}g|, 1 \rangle$ as an extra point representing the non-input ordinal $|\hat{I}g|$ and the initial state, which would otherwise be omitted, and denote the set $(M_0 \alpha)g \in \mathcal{P}(\mathbb{N}^2)$ as given by

$$M_0 = \lambda \alpha. \lambda g. (\lambda s. s \cup (\{|\hat{I}g|\}^1 \parallel (\{1\} - \hat{S}g^1)) \bigcup_{(m,e) \in g} (\mu \lambda i. \langle (\alpha \uparrow \hat{I}g)^{-1} i, m \rangle) \cup \mathcal{D}(\mathcal{D}(e))).$$

Words Part of the problem of identifying an edge in the transducer with a pair of words for the transcoder is now readily solved by identifying each point $p \in (M_0 \alpha) g$ with a pair whose input word is the unit set $\{((M_0 \alpha) g)^\circ p\}$, a well defined value consistent with the output terminal ordering on the transition table block. However, a corresponding output word remains to be expressed. Usually the output word depends on an edge $((i, o), n) \in e$ in the adjacency set $e = (\Psi g) p_1$ whose input burst i contains the input symbol whose ordinal is p_0 . There can be more than one such edge if the process specification is non-deterministic, giving rise to multiple pairs with the same input word, which Equation 13.22 allows in a transcoder specification. Usually the set of these edges would be

$$(\{(\alpha \uparrow \hat{I} g)_{p_0}\}) \times \mathcal{P}(\mathbb{T}) \times \mathbb{N} \cap e$$

when p_1 is a member of $\hat{S} g$. In the unusual case of a non-quiescent initial state $p_1 = 1 \notin \hat{S} g$, there is an output word corresponding to every edge $((\emptyset, o), n) \in e$. The result in either case is expressible as $((M_1 \alpha) g) p$ for M_1 defined as follows.

$$M_1 = \lambda \alpha. \lambda g. \lambda p. (\{(\lambda i. \langle \emptyset, \{(\alpha \uparrow \hat{I} g)_{p_0}\}_i) \delta_{\emptyset}^{\{p_1\} - \hat{S} g}\} \times \mathcal{P}(\mathbb{T}) \times \mathbb{N} \cap (\Psi g) p_1$$

Fields A transcoder specification associating an input word $\{((M_0 \alpha) g)^\circ p\}$ with an output word derived from a list $\langle n, p_0, o \rangle$ for every edge $((i, o), n) \in ((M_1 \alpha) g) p$ would now seem a step closer, because it has the correct next state n , input p_0 , and outputs o to make up the fields on the bus shown in Figure 15.9, but some technicalities remain.



- The next state n must be suppressed if it is not a member of $\hat{S} g$ because there is no column for it in the transition table, and if it is a member of $\hat{S} g$ then it must be transformed to the column number $(\hat{S} g)^\circ n$.
- Similarly, the input p_0 must be suppressed if its not acknowledgeable, meaning i is not a subset of $\hat{A} g$, and must be transformed to its ordinal with respect to those of the rest of the acknowledgeable inputs otherwise.
- The output symbols in o must be converted to their ordinals with respect to $\alpha \uparrow \hat{O} g$.
- Both the input p_0 and the ordinals of the outputs in o must be offset by the widths of the preceding fields on the bus.

An output word containing the subset $(\mu (\hat{S} g)^\circ) (\{n\} \cap \hat{S} g)$ in place of n addresses the first point, also expressible as $(f \hat{S} g) n$ for

$$f : \mathcal{P}(\mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}))$$

defined temporarily as $f = \lambda h. \lambda j. (\mu h^\circ) (\{j\} \cap h)$. The second point is remedied similarly by

$$f((\mu (\alpha \uparrow \hat{I} g)^{-1}) \hat{A} g) p_0$$

and the third by rewriting o to $(\mu (\alpha \uparrow \hat{O} g)^{-1}) o$ in the result $((M_2 \alpha) g) \triangle \langle n, p_0, o \rangle \in \mathcal{P}(\mathbb{N})^3$ by Equation 12.3 with M_2 defined by

$$M_2 = \lambda \alpha. \lambda g. (\lambda f. \langle f \hat{S} g, f((\mu (\alpha \uparrow \hat{I} g)^{-1}) \hat{A} g), \mu (\alpha \uparrow \hat{O} g)^{-1} \rangle) \lambda h. \lambda j. (\mu h^\circ) (\{j\} \cap h).$$

The required offsets noted in the last point above and the union of the three fields to a single output word are effected by writing

$$\bigcup \mathcal{R}((M_3 g) \triangle ((M_2 \alpha) g) \triangle \langle n, p_0, o \rangle) \in \mathcal{P}(\mathbb{N})$$

with the list of three functions $(M_3 g) \in (\mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N}))^3$ defined according to

$$M_3 = \lambda g. (\lambda w. \mu \lambda t. w + t)^* \langle 0, |\hat{S} g|, |\hat{S} g| + |\hat{A} g| \rangle$$

offsetting the second and third fields respectively by $|\hat{S} g|$ and $|\hat{S} g| + |\hat{A} g|$. With both the input and the output word in each pair required by the transcoder specification derived above, an expression for the whole transcoder amounts to

$$t = \text{TC } M_4(\langle M_0 \alpha, M_1 \alpha, M_2 \alpha, M_3 \rangle \triangle g^4) \in \mathbb{H} \quad (15.20)$$

in terms of a function M_4 combining each input word with its respective output words based on a definition

$$M_4 = \lambda m. \bigcup_{p \in m_0} \{ \{ m_0^\circ p \} \} \times (\mu \lambda((i, o), n). \bigcup \mathcal{R}(m_3 \triangle m_2 \triangle \langle n, p_0, o \rangle)) m_1 p.$$

Initialization While [Equation 15.20](#) defines the whole transcoder, it does not necessarily conclude the construction of the controller block. As noted previously, initially non-quiescent processes require special provisions. The extra member $\langle |\hat{I} g|, 1 \rangle \in (M_0 \alpha) g$ in the case of an initially non-quiescent process, being its lexicographic maximum, creates an extra terminal reserved for this situation in the last position on the transcoder's input bus. An input signal applied to this terminal causes the transcoder to emit the output signals o associated with an arbitrarily selected outgoing edge $((\emptyset, o), n)$ from the initial state and to enable the column numbered $(\hat{S} g)^\circ n$ on the transition table. Because this turn of events should take place when the circuit is powered up with no prompting from the environment, it is appropriate to drive the last input terminal on the transcoder with a PUSH, even if there is nothing in particular to drive the PUSH. A circuit of this form derived from the transcoder t is given by

$$\mathbf{L}\langle \mathbf{Z}^2\mathbf{R}(\text{FORK}, \text{PUSH}), t \rangle$$

where one of the outputs from the FORK feeds back to the input to avoid letting it float. Formally mapping a transcoder to a controller block by a transformation $\text{CTLI } g : \mathbb{H} \rightarrow \mathbb{H}$ analogous to [Equation 15.17](#) entails a definition along the lines of

$$\text{CTLI} = \lambda g. \lambda t. (\lambda k. \langle t, \mathbf{L}\langle \mathbf{Z}^2\mathbf{R}(\text{FORK}, \text{PUSH}), t \rangle \rangle_k) \delta_{\emptyset}^{\{1\} - \hat{S} g}$$

which leaves the parameter t representing the transcoder by [Equation 15.20](#) unchanged if the specification is initially quiescent as indicated by the condition $1 \in \hat{S} g$. With that we can write the complete definition of the controller block as

$$\text{CTL} = \lambda \alpha. \lambda g. (\text{CTLI } g) \text{TC } M_4(\langle M_0 \alpha, M_1 \alpha, M_2 \alpha, M_3 \rangle \triangle g^4). \quad (15.21)$$

15.3.3 Combining form

Putting the three building blocks together into the circuit as shown in Figure 15.7 is fairly straightforward aside from providing for an anomaly noted below. A serializer SE b by Equation 15.14 and a transition table (TT α) g by Equation 15.18 combine into a block

$$\mathbf{F}_{|\hat{I}g|} \langle \text{SE } b, (\text{TT } \alpha) g \rangle$$

which combines in turn with a controller (CTL α) g by Equation 15.21

$$u = \mathbf{C}_{\hat{T}g} \langle \mathbf{F}_{|\hat{I}g|} \langle \text{SE } b, (\text{TT } \alpha) g \rangle, (\text{CTL } \alpha) g \rangle \quad (15.22)$$

in terms of a bus width $\hat{T}g \in \mathbb{N}$ from the transition table to the controller given by a summation

$$\hat{T}g = \lambda g. \sum_{m \in \hat{S}g} |(\Psi g) m|$$

of the adjacency set cardinality $|(\Psi g) m|$ of each state $m \in \hat{S}g$, this being the number of outputs from the decision wait in the column numbered $(\hat{S}g)^\circ m$. The feedback paths from the controller to the serializer and the transition table are buses of widths $|\hat{A}g|$ and $|\hat{S}g|$ respectively given by Equation 15.19 and Equation 15.15, so that the whole combination folds together into $(M_5 g) u$ as given by

$$M_5 = \lambda g. \lambda u. (\mathcal{F}_u \lambda(l, r). \mathbf{Z}^l ((\mathbf{Z}^l \mathbf{R}(r, l^l)) \downarrow l)) \langle |\hat{A}g|, |\hat{S}g| \rangle$$

connecting the first output bus to the last input bus twice on the block u derived in Equation 15.22 above. We can summarize this combination as

$$x = ((M_6 M_5) g) (\text{SE } b, \text{TT } \alpha, \text{CTL } \alpha) \quad (15.23)$$

with M_6 defined by

$$M_6 = \lambda m. \lambda g. \lambda(s, t, c). (m g) \mathbf{C}_{\hat{T}g} \langle \mathbf{F}_{|\hat{I}g|} \langle s, t g \rangle, c g \rangle$$

leading to a satisfactory definition of a combining form

$$\Omega_m^\alpha = \lambda d. \lambda g. ((M_6 M_5) g) (\text{SE } d g, \text{TT } \alpha, \text{CTL } \alpha) \quad (15.24)$$

whereby a state based synthetic implementation $(\Omega_m^\alpha \mathcal{U}_m^\alpha) g \in \mathbb{H}$ is expressible in terms of the decomposition given by Equation 15.12 and a feedback anti-refined transducer $g = \mathbf{FAT } X$ for a process specification $X \in \mathbb{D}$.

15.3.4 Loose ends

A circuit of the form $(\Omega_m^\alpha \mathcal{U}_m^\alpha) g$ exposes an interface to its environment based on input and output alphabets inferred from the feedback anti-refined transducer $g = \mathbf{FAT } X$ by Equation 15.10 and Equation 15.11, but these alphabets $\hat{I}g$ and $\hat{O}g$ might not match the alphabets I and O of the process $X = (I, O, N)$. One way $\hat{I}g$ could differ from I is by omitting inputs $i \in I - \hat{I}g$ that are mentioned in I but never appear anywhere in the transducer. As explained in Section 5.3.4, these inputs could be due to an error by the designer and are interpreted as input terminals that exist on the circuit but are never safe to use. The analogous error of outputs in $O - \hat{O}g$ indicates that they

never transmit a signal (unless maybe the system diverges due to an unsafe input, in which case all bets are off). A correct implementation of the process X should take these symbols into account if for no other reason than to ensure the errors are noticed.

Another way the feedback anti-refined transducer alphabet might differ from the process alphabet is by the presence of feedback signals created during the course of the transformation described in Section 15.2.2. Each member of $(\hat{I} \text{ FAT } X) - \hat{I} \text{ AT } X$ is a feedback input representing a terminal meant to be connected to one represented by a corresponding member of $(\hat{O} \text{ FAT } X) - \hat{O} \text{ AT } X$ and not exposed to the environment. A correct implementation of the process X must incorporate these connections through a feedback bus to meet the specification.

Feedback bus

With regard to the latter issue, let $\hat{F}X \in \mathcal{P}(\mathbb{G})^2$ denote the feedback inputs and outputs inferred from a process $X \in \mathbb{D}$ in a list of two sets according to

$$\hat{F} = \lambda X. (\lambda f. (f \text{ FAT } X) - f \text{ AT } X)^* \langle \hat{I}, \hat{O} \rangle.$$

To connect the feedback outputs $(\hat{F}X)_1$ to the feedback inputs $(\hat{F}X)_0$ on a synthesized circuit, we need to know the positions of their terminals, which depend on the alphabet ordering. However, an alphabet ordering α for X need not mention any of the feedback symbols in $\bigcup \mathcal{R}(\hat{F}X)$ at all, so we have to improvise an extended alphabet ordering that puts them where we can find them. An alphabet ordering prefixed by the feedback outputs in their lexicographic order

$$(\hat{F}X)_1^{\circ -1}$$

followed by a list excluding the feedback symbols of X in the order originally specified by α

$$\alpha \uparrow \mathbb{T} - \bigcup \mathcal{R}(\hat{F}X)$$

followed by the feedback inputs in reverse lexicographic order

$$(\mathcal{F}_\epsilon \lambda(h, t). t \parallel \langle h \rangle) (\hat{F}X)_0^{\circ -1}$$

as given by $M_7(\alpha, X) \in \mathbb{T}^*$ according to

$$M_7 = \lambda(\alpha, X). b \langle (\hat{F}X)_1^{\circ -1}, \alpha \uparrow \mathbb{T} - \bigcup \mathcal{R}(\hat{F}X), (\mathcal{F}_\epsilon \lambda(h, t). t \parallel \langle h \rangle) (\hat{F}X)_0^{\circ -1} \rangle$$

means a circuit synthesized as

$$(\Omega_m^a \mathcal{U}_m^a) \text{ FAT } X$$

with Ω_m^a and \mathcal{U}_m^a notably parameterized by $a = M_7(\alpha, X)$ instead of α , has an input bus arranged with the ordinary input symbols preceding the feedback input symbols in reverse order, and an output bus carrying the feedback output symbols followed by the ordinary output symbols. This arrangement makes it easy to express the synthesized circuit complete with its feedback bus as

$$\mathbf{z}^{|\hat{F}X|_0} (\Omega_m^a \mathcal{U}_m^a) \text{ FAT } X$$

because it has the first $|\hat{F}X|_0 = |\hat{F}X|_1$ outputs on $(\Omega_m^a \mathcal{U}_m^a) \text{ FAT } X$ connected the last inputs on it in the right order to match each feedback output with the corresponding feedback input.

Useless terminals

To complete the synthesized circuit with additional terminals accounting for any alphabet symbols not appearing in the transducer, we could make a list of three sets

$$l = \langle I, \hat{I} g, (\hat{I} \mathbf{FAT} X) - \hat{I} \mathbf{AT} X \rangle \in \mathcal{P}(\mathbb{T})^3$$

in reference to a process $X = (I, O, N) \in \mathbb{D}$ and a transducer $g = \mathbf{FAT} X$ so that the two-item list

$$\langle l_1 - l_2, l_0 - l_1 \rangle = \langle (\hat{I} g) - ((\mathbf{FAT} X) - \hat{I} \mathbf{AT} X), I - \hat{I} g \rangle \in \mathcal{P}(\mathbb{T})^2$$

contains the set of useful (non-feedback) input alphabet symbols as its first term and the set of unsafe or useless inputs as its second term. A list of analogous results for both input and output alphabet symbols

$$(\lambda l. \langle l_1 - l_2, l_0 - l_1 \rangle)^* \langle \langle I, O \rangle, \langle \hat{I} g, \hat{O} g \rangle, \hat{F} X \rangle^\top \in (\mathcal{P}(\mathbb{T})^2)^2$$

in combination with an alphabet ordering α for X indicates the terminal numbers

$$(\lambda l. (\mu (\alpha \uparrow l_0))^{-1} \langle l_1 - l_2, l_0 - l_1 \rangle)^* \langle \langle I, O \rangle, \langle \hat{I} g, \hat{O} g \rangle, \hat{F} X \rangle^\top \in (\mathcal{P}(\mathbb{N})^2)^2$$

required for each set of symbols on the resulting circuit, expressible more succinctly as $M_8(X, g) \alpha$ in terms of a function

$$M_8 = \lambda(X, g). \lambda((I, O, N). \lambda \alpha. (\lambda l. (\mu (\alpha \uparrow l_0))^{-1} \langle l_1 - l_2, l_0 - l_1 \rangle)^* \langle \langle I, O \rangle, \langle \hat{I} g, \hat{O} g \rangle, \hat{F} X \rangle^\top) X.$$

In other words, for a list $m = M_8(X, g) \alpha \in (\mathcal{P}(\mathbb{N})^2)^2$, we have useful input ordinals $m_{00} \in \mathcal{P}(\mathbb{N})$, useless inputs m_{01} , useful outputs m_{10} , and useless outputs m_{11} .

Enhancing any circuit $x \in \mathbb{H}$ with $|m_{01}|$ additional useless inputs and $|m_{11}|$ additional useless outputs is a straightforward matter of embedding it in a booby trap

$$(\mathcal{F} \mathbf{R}) \langle x, (\mathbf{Z}^2 \mathbf{R}(\mathbf{PUSH}, \mathbf{JOIN}))^{|m_{01}|}, (\mathbf{Z} \mathbf{FORK})^{|m_{11}|} \rangle \in \mathbb{H}$$

which diverges when a signal is received on any of its last $|m_{01}|$ input terminals, and otherwise never transmits a signal on any of its last $|m_{11}|$ outputs (cf. [Figure 11.1](#) and [Figure 15.3](#)). If we have

$$x = \mathbf{Z}^{(|\hat{F} X|_0)} (\Omega_m^a \mathcal{U}_m^a) \mathbf{FAT} X$$

with $a = M_7(\alpha, X)$ as proposed above, then $(\overset{\circ}{b} m_0)^{-1}$ and $(\overset{\circ}{b} m_1)^{-1}$ with $m = M_8(X, g) \alpha$ specify the permutation networks by [Equation 11.2](#) needed for the block $(M_9 m) x$ to implement X under an alphabet ordering α based on a definition of

$$M_9 = \lambda m. \lambda x. (\overset{\circ}{b} m_0)^{-1} \times (\mathcal{F} \mathbf{R}) \langle x, (\mathbf{Z}^2 \mathbf{R}(\mathbf{PUSH}, \mathbf{JOIN}))^{|m_{01}|}, (\mathbf{Z} \mathbf{FORK})^{|m_{11}|} \rangle \times (\overset{\circ}{b} m_1)^{-1}. \quad (15.25)$$

Hence it might be helpful to define a comprehensive basic state based synthesis algorithm in terms of a function $\text{SBS}_0 : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$ given by

$$\text{SBS}_0 = \lambda(\alpha, X). (\lambda(a, g). (M_9 M_8(X, g) \alpha) \mathbf{Z}^{(|\hat{F} X|_0)} (\Omega_m^a \mathcal{U}_m^a) g) (M_7(\alpha, X), \mathbf{FAT} X). \quad (15.26)$$

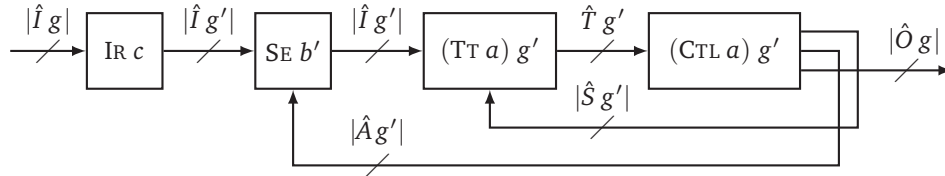
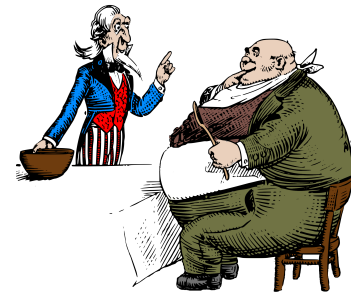


Figure 15.10: Synthesis of a transducer g with input reduction depends on the decomposition $(c, g') = \mathcal{U}_m^\alpha g$, an alphabet ordering $a = (\hat{I} g')^{\circ -1} \parallel \alpha$, and an array of $|\hat{I} g'|$ completion detectors.

15.4 Input reduction

Although the method proposed in the Section 15.3 is adequate to implement any specification, it neglects a possible optimization in examples such as these.



- If two inputs a and b never appear together in the same input burst, and substituting each with the other in every input burst throughout the transducer leaves it invariant, then it would be better to put both a and b through a front end MERGE and treat the output of the MERGE as a single input as far as the rest of the implementation is concerned.
- If an input c always appears in any input burst containing an input d and *vice versa*, then a similar optimization is appropriate with a JOIN instead of a MERGE.
- If every input burst containing any of e, f , and g always contains exactly two of them and each pair is always interchangeable with the others as above, then all three should be fed through a 2-of-3 completion detector whose output is treated as a single input thereafter.

Each of these optimizations requires some additional hardware on the front end, but is likely to be a net improvement because it results in at least one less row on the decision wait in the transition table block and at least one less input to the controller. The latter two examples also simplify the serializer by reducing or eliminating a sequencer, and the first example may do so as well if either a or b can be concurrent with other inputs.

These optimizations need not depend on fortuitous observations, but can be recognized and applied systematically. In every case, there is a set of related inputs combined into one component in the front end, and if we construct the intersection of this set with every input burst that intersects it in the transducer, the result is an antichain. This antichain can be treated as a delay insensitive code amenable to completion detection as explained in Section 13.4. Even a MERGE or a JOIN is a special case of a completion detector.

A suitable analysis enables the implementation of a transducer g by an array $(\mathcal{F} \mathbf{R}) \text{CD}^* \eta c$ of completion detectors labeled IR c in Figure 15.10 on the front end, and a back end implementing a transducer g' of possibly lower input arity $|\hat{I} g'| \leq |\hat{I} g|$ as in Figure 15.7. The analysis yields the list of antichains $c \in \mathcal{P}(\mathcal{P}(\mathcal{R}(\iota_{|\hat{I} g|})))^*$ to specify the completion detectors when renumbered by

Equation 13.16. A permutation network not shown concludes the construction by connecting each input terminal numbered $i = (\alpha \uparrow \hat{I} g)^{-1} a$ of an alphabet symbol $a \in \hat{I} g$ to the completion detector associated with it, which is j -th in the array where i is a member of $\bigcup c_j$.

This solution is split similarly to previous ones between a decomposition and a combining form. The decomposition

$$\mathcal{U}_m^\alpha g = (c, g') \quad (15.27)$$

transforming the transducer g into the list of antichains c discussed above and the transducer g' of reduced input arity is derived in [Section 15.4.1](#). The combining form Ω_m^α derived in [Section 15.4.2](#) provides for an expression of the circuit in [Figure 15.10](#) as $\Omega_m^\alpha \mathcal{U}_m^\alpha g$.

15.4.1 Decomposition

The derivation of the decomposition is lengthy enough by itself to need a few steps with some explanation for each. A notion of interchangeable input bursts leads to the ensemble of delay insensitive codes determining the front end array of completion detectors. This partial result then suggests an algorithm for rewriting the alphabet of the transducer accordingly.

Interchangeable inputs

If some subset $i \in \mathcal{P}(\mathbb{T})$ of the input alphabet is interchangeable with some other subset $j \in \mathcal{P}(\mathbb{T})$ in that a single input could substitute for both, it is still neither feasible nor worthwhile to make the substitution unless no members of i or j are left over in the transducer afterwards. Some members of i or j could be left over if they appear in an input burst that does not contain the rest of them. A simple algorithm for finding usefully interchangeable input subsets by these criteria follows by substituting one for the other in every input burst $k \in \mathcal{D}(\mathcal{D}(g))$ of a transducer g where it is present or rewriting k to exclude its members where it is not, and then checking that the transducer is invariant with respect to this operation. The specific effect on an input burst k is achieved by unpacking it into its subsets $l \in \mathcal{P}(k)$ and then repacking them into a set

$$\bigcup_{l \in \mathcal{P}(k)} \langle \langle l - (i \cup j), i \rangle_{\delta_i^l}, j \rangle_{\delta_j^l}$$

such that either of i or j is exchanged for the other where applicable and suppressed otherwise. The effect on an adjacency set e due to this transformation of every input burst k associated with every edge $((k, o), n) \in e$ is a rewritten adjacency set

$$\bigcup_{((k,o),n) \in e} \{ \langle \langle \bigcup_{l \in \mathcal{P}(k)} \langle \langle l - (i \cup j), i \rangle_{\delta_i^l}, j \rangle_{\delta_j^l}, o \rangle, n \rangle \}.$$

To put this idea to work, let i and j be members of the set

$$s = \bigcup_{b \in \mathcal{D}(\mathcal{D}(e))} \mathcal{P}(b) \in \mathcal{P}(\mathcal{P}(\mathbb{T}))$$

of subsets of acceptable input bursts b according to an adjacency set e , and form the partition on s

$$(\pi \lambda i. \{j \in s \mid e = \bigcup_{((k,o),n) \in e} \{ \langle \langle \bigcup_{l \in \mathcal{P}(k)} \langle \langle l - (i \cup j), i \rangle_{\delta_i^l}, j \rangle_{\delta_j^l}, o \rangle, n \rangle \} \}) s \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{T})))$$

by Equation 6.6 such that any two input burst subsets $i, j \in s$ belong to the same class if and only if rewriting e as explained above with respect to i and j makes no difference to it. Then we have in $d_0 g \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{T}))))$ the set of all such partitions throughout g as given by

$$d_0 = \lambda g. \bigcup_{(m,e) \in g} \{ (\lambda s. (\pi \lambda i. \{j \in s \mid e = \bigcup_{((k,o),n) \in e} \{ \langle \langle l - (i \cup j), i \rangle_{\delta_i^l}, j \rangle_{\delta_i^l}, o \rangle, n \rangle \} \}) s) \bigcup_{b \in \mathcal{D}(\mathcal{D}(e))} \mathcal{P}(b) \}. \quad (15.28)$$

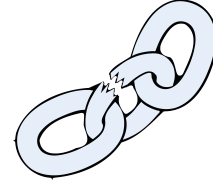
That is, each member $v \in d_0 g$ is a partition by interchangeability, each equivalence class $t \in v$ is a set of input bursts, and each input burst $i \in t$ is a set of input alphabet symbols. It may be helpful to think of every class t as an antichain because input bursts are not normally interchangeable with proper subsets of themselves (or else an input in them is ignored the first time it arrives and causes divergence the second time), but this assumption is neither necessary nor fully reliable. To provide for pathological cases, we can transform $d_0 g$ to $d_1 d_0 g$ by

$$d_1 = \lambda w. \bigcup_{v \in w} \bigcup_{t \in v} \mathcal{R}((\lambda(h : z). (\lambda l. \langle l : h - l : z, h : z \rangle_{\delta_i^l}) (\bigcup_{i \in h} \mathcal{P}(i) - h))^\infty \langle t \rangle) \quad (15.29)$$

which separates the lower links $l \subset h$ from any chain $h \subseteq t$ present in a class t successively until t is separated into as many subsets as needed to ensure that each subset is an antichain. Ensuring that each class is an antichain justifies treating it as a delay insensitive code from which to derive a completion detector (Section 13.1.4).

This formulation makes some progress towards identifying all interchangeable input bursts, but is not quite the solution. Notably the membership of two input bursts $i, j \in t$ in the same class $t \in v$ of some partition $v \in d_1 d_0 g$ implies only that there exists a state $m \in \mathcal{D}(g)$ wherein they are interchangeable, not that they are interchangeable unconditionally. Recall that the goal of this analysis is to optimize the circuit by feeding all members of any interchangeable input sets i and j through the same completion detector on the front end. This optimization would not be valid unless they were interchangeable in every state.

Redefining d_0 in terms of a union $e = \bigcup \mathcal{R}(g)$ of all adjacency sets in the transducer g instead of examining them individually would resolve this issue but would be suboptimal. Sets i and j of inputs that are interchangeable in a state m (because the same output burst o and succeeding state n follow in either alternative) can also be interchangeable in another state m' though they might then lead to a different output burst o' and succeeding state n' . However, because the transducer can occupy only one state at a time, it is correct to regard i and j as interchangeable regardless. We would miss an opportunity for optimization by insisting on identical output bursts and succeeding states globally, but this consequence would be inevitable without analyzing each adjacency set individually as above.



Compatible partitions

Nevertheless, sticking with the given formulation admits incompatible partitions $v \in d_1 d_0 g$ needing some sort of resolution whenever two bursts are interchangeable according to one but not another. Occasionally this effect is due to partitions being disjoint from each other because their inputs are never acceptable simultaneously in the same state. A reasonable course in this situation would be to treat all equivalences implied by each partition as generally applicable in all states, because in states where some of the inputs are prohibited, they either never arrive or relieve the implementation of

any obligations when they do. Disjoint partitions $u, v \in d_1 d_0 g$ whose classes contain none of the same input bursts and whose input bursts contain none of the same inputs might just as well be the single partition $u \cup v$.

Occasionally the situation is more complicated because separate partitions can imply different conditions about input bursts containing the same symbols. If there is a partition v containing a class $t = \{i, j\} \in v$ and another partition u containing a class $s = \{j, k\} \in u$, then for similar reasons as above, it is appropriate to treat all three input bursts i, j , and k as mutually interchangeable in every state even though i and k are prohibited in some. Generalizing from this example would suggest a method of reconciling the two partitions by merging any classes that intersect. However, this method fails on the example of

$$v = \{\{i, j\}, \{k, l\}\} \quad (15.30)$$

$$u = \{\{i, k\}, \{j, l\}\} \quad (15.31)$$

because it leads to a combined partition $\{\{i, j, k, l\}\}$ asserting that every input burst is interchangeable with every other, when clearly neither of k or l is interchangeable with i or j in the state from which v has been inferred.

A better idea is to merge every class $t \in v$ with every class $s \in u$ that intersects t and crucially does not intersect any other class in v . To make this idea more precise, we can express the set

$$u - \mathcal{P}((\bigcup u) - t)$$

of classes in u that intersect t , the set

$$u - \mathcal{P}((\bigcup u) - \bigcup(v - \{t\}))$$

of classes in u that intersect classes other than t in v , and therefore the set

$$(u - \mathcal{P}((\bigcup u) - t)) - (u - \mathcal{P}((\bigcup u) - \bigcup(v - \{t\})))$$

of all classes $s \in u$ that should be merged with t . Then the combined partition contains all classes in

$$(\mu \lambda t. \bigcup_{s \in (u - \mathcal{P}((\bigcup u) - t)) - (u - \mathcal{P}((\bigcup u) - \bigcup(v - \{t\})))} t \cup s) \nu$$

but this set is empty if the alphabet $\bigcup \bigcup u \subset \mathbb{T}$ of the partition u is disjoint from that of v as in the simpler case considered above. To cover both cases, classes $t \in v$ whose input bursts $i \in t$ contain no members of $\bigcup \bigcup u$ should also be included in the result. These classes would be those members of v that are also members of the set

$$\mathcal{P}(\mathcal{P}(\mathbb{T} - \bigcup \bigcup u))$$

of all sets of input bursts made of non-members of the alphabet of u , or more succinctly $v \cap f u$ for

$$f = \lambda w. \mathcal{P}(\mathcal{P}(\mathbb{T} - \bigcup \bigcup w)).$$

Similarly, members of u whose input bursts are disjoint from the alphabet of v should be included in the combined result

$$(v \cap f u) \cup (u \cap f v)$$

or more explicitly

$$(\lambda f. \bigcup_{r \in (v \cap f u) \cup (u \cap f v)} \{r\}) \lambda w. \mathcal{P}(\mathcal{P}(\mathbb{T} - \bigcup \bigcup w))$$

along with the results of intersecting classes in

$$p = ((\lambda f. \bigcup_{r \in (v \cap f u) \cup (u \cap f v)} \{r\}) \lambda w. \mathcal{P}(\mathcal{P}(\mathbb{T} - \bigcup \bigcup w))) \cup (\mu \lambda t. \bigcup_{s \in (u - \mathcal{P}((\bigcup u) - t)) - (u - \mathcal{P}((\bigcup u) - \bigcup (v - \{t\})))} t \cup s) v).$$

However, even this result is empty in reference to [Equation 15.30](#) and [Equation 15.31](#) because there is no class $s \in u$ compatible with any class $t \in v$, and every member of v intersects some member of u . The right answer for this example is

$$\bigcup_{a \in \bigcup \bigcup (u \cup v)} \{\{a\}\}$$

because any input $a \in i \cup j \cup k \cup l$ excluded by both criteria belongs in a unit input burst $\{a\}$ in a unit class $\{\{a\}\}$ signifying that the only condition it satisfies in every state is interchangeability with itself. The combined partition associated with any arbitrary partitions $u, v \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{T})))$ taking into account this case as well would be more like

$$(\lambda p. p \cup \bigcup_{a \in (\bigcup \bigcup (u \cup v)) - \bigcup \bigcup p} \{\{a\}\}) ((\lambda f. \bigcup_{r \in (v \cap f u) \cup (u \cap f v)} \{r\}) \lambda w. \mathcal{P}(\mathcal{P}(\mathbb{T} - \bigcup \bigcup w))) \cup (\mu \lambda t. \bigcup_{s \in (u - \mathcal{P}((\bigcup u) - t)) - (u - \mathcal{P}((\bigcup u) - \bigcup (v - \{t\})))} t \cup s) v).$$

Generally there are not just two partitions u and v of input bursts, but up to as many partitions as states in the transducer g , all of which have to be reduced to the common partition describing the front end completion detector stage. A reduction operating on a list of partitions can be obtained as usual by folding a binary operation over the list, and a function

$$d_2 : \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))^* \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$$

defined below operates on a list of partitions wherein the inputs are represented numerically with respect to an alphabet ordering α in preparation for the upcoming definition of \mathcal{U}_m^α , the decomposition function.

$$d_2 = \mathcal{F}_\emptyset \lambda(u, v). (\lambda p. p \cup \bigcup_{a \in (\bigcup \bigcup (u \cup v)) - \bigcup \bigcup p} \{\{a\}\}) ((\lambda f. \bigcup_{r \in (v \cap f u) \cup (u \cap f v)} \{r\}) \lambda w. \mathcal{P}(\mathcal{P}(\mathbb{N} - \bigcup \bigcup w))) \cup (\mu \lambda t. \bigcup_{s \in (u - \mathcal{P}((\bigcup u) - t)) - (u - \mathcal{P}((\bigcup u) - \bigcup (v - \{t\})))} t \cup s) v)$$

For the moment, it can be noted that

$$d_2 ((\mu^4(\alpha \uparrow \hat{I} g)^{-1}) d_1 d_0 g)^{\circ^{-1}} \in \mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{N})))$$

expresses precisely the resulting partition in terms of [Equation 15.28](#), [Equation 15.29](#), the alphabet ordering α , and the transducer g .

Compatible classes

Ordinarily each equivalence class in the partition obtained above corresponds to a separate completion detector in the front end stage of [Figure 15.10](#), a parallel combination of completion detectors with a collective input arity $|\hat{I} g|$ issues from a listing of the classes in any fixed order, and each

input to each completion detector is exposed through a permutation network determined by the order. However, nothing in the derivation up to this point would guarantee an input arity of $|\hat{I}g|$. Although no input burst can appear in more than one class, nothing prevents input bursts in different classes from intersecting. If two classes have intersecting alphabets, the only conclusion is that no input burst in either class is unconditionally interchangeable with any other. This anomaly may be unusual (cf. Equation 15.29), but thoroughness requires transforming any classes affected by it to a set of unit classes of unit input bursts as explained above.

The transformation is relevant to a partition v , a class $t \in v$, the set $\mathbb{N} - \bigcup t$ of inputs other than those appearing in any member of t , the set $\mathcal{P}(\mathbb{N} - \bigcup t)$ of input bursts disjoint from any member of t , and the set $\mathcal{P}(\mathcal{P}(\mathbb{N} - \bigcup t))$ containing all possible classes in which every input burst is disjoint from any member of t . If the set

$$v - \mathcal{P}(\mathcal{P}(\mathbb{N} - \bigcup t))$$

of classes in v whose alphabets are not disjoint from that of t contains anything other than t , then every member r of

$$w = \bigcup_{t \in v} \{t\} \cup \bigcup_{s \in v - \mathcal{P}(\mathcal{P}(\mathbb{N} - \bigcup t))} \{s\}$$

should be rewritten in v to the set of classes

$$\bigcup_{a \in \bigcup \bigcup r} \{\{\{a\}\}\}$$

obtained by transforming any input a in any member of any class in r to a class $\{\{a\}\}$ containing only the input burst $\{a\}$. We can summarize this transformation as $d_3 v \in \mathcal{P}(\mathcal{P}(\mathbb{N}))^*$ with a function d_3 defined by

$$d_3 = \lambda v. ((\lambda w. \bigcup_{r \in w} \langle r, \bigcup_{a \in \bigcup \bigcup r} \{\{\{a\}\}\} \rangle_{1-\delta_1^{|r|}}) (\bigcup_{t \in v} \{t\} \cup \bigcup_{s \in v - \mathcal{P}(\mathcal{P}(\mathbb{N} - \bigcup t))} \{s\}))^{\circ-1}$$

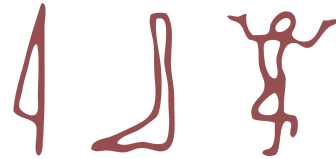
also taking the opportunity to list the classes in lexicographic order. Then the result

$$c = d_3 d_2 ((\mu^4 (\alpha \uparrow \hat{I}g)^{-1}) d_1 d_0 g)^{\circ-1} \quad (15.32)$$

fully constrains both the completion detector array and the input permutation network, concluding the derivation of the left side of the decomposition $(c, g') = \mathcal{U}_m^\alpha g$ sought in Equation 15.27.

Rewritten alphabet

The rest of the decomposition is about deriving a transducer g' with a reduced input alphabet from the transducer g consistent with the codes ηc by Equation 15.32 and Equation 13.16. The simplest approach to transforming the alphabet is to seek a non-injective map from input symbols of g to input symbols of g' such that the index n in the list c of the term c_n whose union $\bigcup c_n$ contains the ordinal $i = (\alpha \uparrow \hat{I}g)^{-1} a$ of an input symbol $a \in \hat{I}g$ associates a with the n -th member of $\hat{I}g'$. The n -th member of $\hat{I}g$ can be made precise by drawing $\hat{I}g'$ from from the universe $\mathbb{G} \subset \mathbb{T}$ of generic symbols defined in Section 8.7.1



and relying on its usual ordering. A generic symbol $\mathbb{G}^{\circ-1} n$ substituted throughout g based on the value of n obtained by

$$n = c^{-1} (\Psi \bigcup_{k \in \mathcal{D}(c)} (\bigcup c_k) \times \{c_k\}) i$$

for each ordinal i would result in a transducer with an input alphabet of $|c|$ symbols having fused any subsets of inputs to a common completion detector into a single input. However, to maintain the condition that the input alphabet of a transducer should be disjoint from the output alphabet, it would be better to offset each value of n by a value

$$1 + \max(\{0\} \cup (\mu \mathbb{G}^{\circ}) (\mathbb{G} \cap \hat{O} g))$$

exceeding the ordinal of the maximum generic symbol in the output alphabet $\hat{O} g$, if any, for a transformed input symbol $d_4(c, i, \hat{O} g) \in \mathbb{G}$ given by

$$d_4 = \lambda(c, i, o) \cdot \mathbb{G}^{\circ-1} (1 + \max(\{0\} \cup (\mu \mathbb{G}^{\circ}) (\mathbb{G} \cap o)) + c^{-1} (\Psi \bigcup_{k \in \mathcal{D}(c)} (\bigcup c_k) \times \{c_k\}) i).$$

Modified transducer

This transformation applied in the context of an input burst b labeling an edge $((b, o), n) \in e$ in the adjacency set e of a vertex $(m, e) \in g$ of a transducer g leads to an input burst

$$\bigcup_{i \in (\mu(\alpha \uparrow \hat{I} g)^{-1}) b} \{d_4(c, i, \hat{O} g)\}$$

resulting in a rewritten adjacency set

$$\bigcup_{((b,o),n) \in e} \{(\bigcup_{i \in (\mu(\alpha \uparrow \hat{I} g)^{-1}) b} \{d_4(c, i, \hat{O} g)\}, o), n\}$$

in every vertex of the transducer

$$g' = \bigcup_{(m,e) \in g} \{(m, \bigcup_{((b,o),n) \in e} \{(\bigcup_{i \in (\mu(\alpha \uparrow \hat{I} g)^{-1}) b} \{d_4(c, i, \hat{O} g)\}, o), n\})\}.$$

With both components derived above, we have now the complete decomposition $(c, g') = \mathcal{U}_m^\alpha g$ given by

$$\mathcal{U}_m^\alpha = \lambda g. (\lambda c. (c, \bigcup_{(m,e) \in g} \{(\bigcup_{((b,o),n) \in e} \{(\bigcup_{i \in (\mu(\alpha \uparrow \hat{I} g)^{-1}) b} \{d_4(c, i, \hat{O} g)\}, o), n\})\})) d_3 d_2 ((\mu^4(\alpha \uparrow \hat{I} g)^{-1}) d_1 d_0 g)^{\circ-1}. \quad (15.33)$$

15.4.2 Combining form

As noted previously, this decomposition is constructed to provide for a front end block $(\mathcal{F} \mathbf{R}) \text{CD}^* \hat{\eta} c$ of output arity $|c|$ and a transducer g' with an input alphabet of a size to match, but an input permutation network is needed to interface the front end with the external environment if the original specification is to be met. To this end, we recall that each term $t \in \mathcal{R}(c)$ reflects a completion

detector connected to the externally visible terminals numbered within $\bigcup t \in \mathcal{P}(\mathcal{R}(t_{|\hat{I}g|}))$. Hence the permutation

$$p = \flat(\lambda t. (\bigcup t)^{\circ-1})^* c \in \mathcal{R}(t_{|\hat{I}g|})^*$$

lists for each completion detector input terminal the ordinal of the external terminal to which it should be wired, and does so in order of the positions of the completion detector input terminals on the array $(\mathcal{F} \mathbf{R}) \text{CD}^* \hat{\eta} c$. By convention, the inverse p^{-1} specifies the appropriate input permutation network in the definition of

$$\text{IR} = \lambda c. ((\flat(\lambda t. (\bigcup t)^{\circ-1})^* c))^{-1} \times (\mathcal{F} \mathbf{R}) \text{CD}^* \hat{\eta} c.$$

Figure 15.10 indicates a back end of the form $(\Omega_m^\alpha \mathcal{U}_m^\alpha) g'$ by Equation 15.12 and Equation 15.24 in terms of the transducer g' from the decomposition $(c, g') = \mathcal{U}_m^\alpha g$, except that an alphabet ordering α for g may be inadequate for g' because their input alphabets differ. Unlike the input alphabet of g , the input alphabet $\hat{I} g'$ contains $|c|$ generic symbols of which the lexicographically n -th symbol is associated with the n -th completion detector output from $\text{IR} c$. The simplest choice of an alphabet ordering for g' would be

$$a = (\hat{I} g')^{\circ-1} \parallel \alpha$$

based on the alphabet ordering α of g , because then the back end $(\Omega_m^a \mathcal{U}_m^a) g'$ would be synthesized with input terminals ordered to match the outputs from the front end, and the two could be connected in a cascade

$$\mathbf{F}_{|c|} \langle \text{IR} c, (\Omega_m^a \mathcal{U}_m^a) g' \rangle$$

with no permutation network needed between them. The outputs from the cascade would also be ordered the same as those of g would be under α . A combining form following directly as

$$\Omega_m^\alpha = \lambda(c, g'). (\lambda a. \mathbf{F}_{|c|} \langle \text{IR} c, (\Omega_m^a \mathcal{U}_m^a) g' \rangle) ((\hat{I} g')^{\circ-1} \parallel \alpha) \quad (15.34)$$

enables the complete expression of the circuit shown in Figure 15.10 as $\Omega_m^\alpha \mathcal{U}_m^\alpha g$ in terms of a given transducer g and alphabet ordering α .

Whereas $\Omega_m^\alpha \mathcal{U}_m^\alpha g$ is always a compatible replacement for $(\Omega_m^\alpha \mathcal{U}_m^\alpha) g$, it entails the same limitations noted in Section 15.3.4 if the transducer $g = \mathbf{FAT} X$ does not fully capture the process semantics $X \in \mathbb{D}$ because of unused inputs and outputs or because of feedback signals created to cope with non-deterministic concurrency. Fortunately there is no need to retrace the ensuing line of reasoning, but simply to define the transformation $\text{SBS}_1 : (\mathbb{T}^* \times \mathbb{D}) \rightarrow \mathbb{H}$ analogously to Equation 15.26.

$$\text{SBS}_1 = \lambda(\alpha, X). (\lambda(a, g). (M_9 M_8(X, g) \alpha) \mathbf{Z}^{(\hat{F} X)_0} \Omega_m^a \mathcal{U}_m^a g) (M_7(\alpha, X), \mathbf{FAT} X) \quad (15.35)$$

15.5 State reduction

A simple *ad hoc* optimization to the implementation is possible when the transducer model of a process specification has only a single state and its input bursts form an antichain. Despite the restriction to a single transducer state, this class of processes is fairly broad, encompassing all decision waits and sparse decision waits of any dimensions or coordinates, as well as all encoders, decoders, transcoders, majority gates, and completion detectors for any delay insensitive code. In

this case, the serializer, input reducer, transition table, and controller blocks needed for multi-state transducers as shown in [Figure 15.10](#) can all be eliminated in favor of a single transcoder. The transcoder is likely to be a more efficient implementation, especially if the relevant code is factorable or separable ([Section 13.3.3](#) and [Section 13.3.4](#)) and well optimized decision wait decomposition strategies are available.

15.5.1 Decomposition

If we assume for the moment that a transducer g is of the form described above, then the adjacency set e of its single state given by

$$e = \bigcup \mathcal{R}(g)$$

determines a set of i/o bursts $(i, o) \in \mathcal{D}(e)$ from which the set of input and output code words

$$((\mu(\alpha \uparrow \hat{I} g)^{-1}) i, (\mu(\alpha \uparrow \hat{O} g)^{-1}) o) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N})$$

according to an alphabet ordering α suffices to specify a transcoder by [Equation 13.22](#) to implement the specification. Let this set be denoted $\mathcal{U}_m^\alpha g \in \mathcal{P}(\mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}))$ in terms of the decomposition function

$$\mathcal{U}_m^\alpha = \lambda g. (\mu \lambda(i, o). ((\mu(\alpha \uparrow \hat{I} g)^{-1}) i, (\mu(\alpha \uparrow \hat{O} g)^{-1}) o)) \mathcal{D}(\bigcup \mathcal{R}(g)). \quad (15.36)$$

15.5.2 Combining form

To confirm that this decomposition is applicable to a given transducer g , we need to verify that no member of the set

$$s = \bigcup_{i \in b} \mathcal{P}(i) - \{i\}$$

containing all proper subsets of input bursts i in the set $b = \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g)))$ of all input bursts associated with g is itself an input burst. This condition is equivalent to

$$s \cap b = \emptyset.$$

We also need to confirm that there is only one state by $|g| = 1$. If these conditions hold, then the implementation is given by Tc $\mathcal{U}_m^\alpha g$ based on [Equation 13.22](#) and [Equation 15.36](#), but can otherwise revert to the input reduced form $\Omega_m^\alpha \mathcal{U}_m^\alpha g$ based on [Equation 15.33](#) and [Equation 15.34](#). In either case, we have

$$(\lambda k. \langle \Omega_m^\alpha \mathcal{U}_m^\alpha g, \text{Tc } \mathcal{U}_m^\alpha g \rangle_k) \delta_1^{|g|} \delta_\emptyset^{s \cap b}$$

or more precisely $(\Omega_m^\alpha \mathcal{U}_m^\alpha) g$ in terms of the combining form

$$\Omega_m^\alpha = \lambda d. \lambda g. (\lambda b. (\lambda s. (\lambda k. \langle \Omega_m^\alpha \mathcal{U}_m^\alpha g, \text{Tc } d g \rangle_k) \delta_1^{|g|} \delta_\emptyset^{s \cap b}) \bigcup_{i \in b} \mathcal{P}(i) - \{i\}) \mathcal{D}(\mathcal{D}(\bigcup \mathcal{R}(g))).$$

The analog to [Equation 15.26](#) and [Equation 15.35](#) for transforming a specification in terms of a process $X \in \mathbb{D}$ to a member of \mathbb{H} with provisions for any misplaced alphabet symbols or feedback signals would be the following.

$$\text{SBS}_2 = \lambda(\alpha, X). (\lambda(a, g). (M_9 M_8(X, g) \alpha) \mathbf{Z}^{|\hat{F}X|_{01}} (\Omega_m^a \mathcal{U}_m^a) g) (M_7(\alpha, X), \mathbf{FAT} X) \quad (15.37)$$

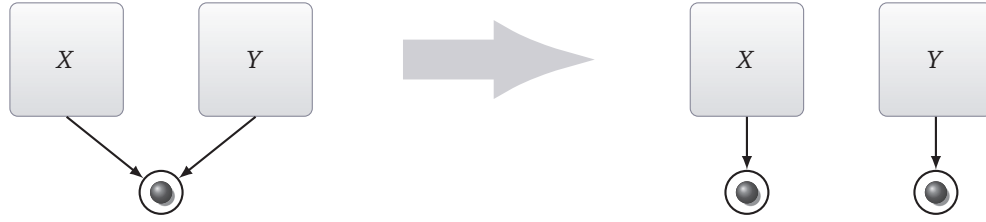


Figure 15.11: Two processes X and Y making up $\text{par}(X, Y)$ can be synthesized separately if their alphabets are disjoint and their Petri net models are either disconnected or have only a dead-end place in common.

15.6 Separation

One further optimization to conclude this chapter on state based synthesis is well worth using on the rare occasions when it applies. If a process is of the form $\text{par}(X, Y)$, then building the result from separate circuits synthesized for X and Y is preferable, and it can be done if their alphabets are disjoint. Not only does the result benefit from smaller decision waits and other building blocks for X and Y than for the combination, but the computational cost of synthesizing it is drastically reduced because the transducers are smaller. As a rough guide, the number $|\mathbf{T} \text{par}(X, Y)|$ of transducer states tends to vary in proportion to the product of $|\mathbf{T} X|$ and $|\mathbf{T} Y|$. The technique also extends to more than two processes with commensurate advantages.

The method involves separating a given process $(I, O, N) \in \mathbb{D}$ into multiple smaller processes by treating its Petri net model N as a graph. Each connected component of the graph N and the observable transitions among its vertices determine the Petri net model and the alphabets of a process to be synthesized separately as a circuit. A parallel combination of the circuits obtained in this way along with input and output permutation networks to maintain a given alphabet ordering implements the overall specification.

A minor exception to the requirement for disjoint components makes the technique a bit more generally applicable. If the Petri net has components that are disjoint except for sharing a common marked or unmarked place with no outgoing arcs, it is valid to regard each component as having its own copy of the shared place as shown in Figure 15.11. This provision does not extend to shared transitions.

Details of this optimization in the rest of this section lead to a state based synthesis method building on the previous ones.

15.6.1 Decomposition

Treating a well formed Petri net $N = (P, T, A, M, F) \in \mathbb{P}$ as an undirected graph, we refer to the adjacency set $f v \in \mathcal{P}(P \cup T)$ of a vertex $v \in P \cup T$ as the union of its postset and preset according to a function

$$f = \Psi \Pi (A \cup (\mu \lambda(i, j). (j, i)) A)$$

and denote by $l = T \cup \mathcal{D}(A)$ the set containing all transitions and all non-terminal places. A partition

$$p = (\pi \rho \lambda v. l \cap f v) \quad l \in \mathcal{P}(\mathcal{P}(l))$$

associates any two vertices in l connected by an undirected path not passing through a terminal place, and any class $c \in p$ extended to

$$c \cup \bigcup_{t \in c} f t$$

includes any terminal places immediately adjacent to its members. The set $s = d_5(T, A)$ given by

$$d_5 = \lambda(T, A). (\lambda(f, l). \bigcup_{c \in (\pi \rho \lambda v. l \cap f v) l} \{c \cup \bigcup_{t \in c} f t\}) (\Psi \Pi (A \cup (\mu \lambda(i, j). (j, i) A), T \cup \mathcal{D}(A))$$

therefore contains connected sets of vertices that are either mutually disjoint or intersect only at terminal places. This set in turn associates a set $(d_6 X) d_5(T, A) \in \mathcal{P}(\mathbb{D})$ of processes with a process $X \in \mathbb{D}$ according to

$$d_6 = \lambda(I, O, (P, T, A, M, F)). \mu \lambda c. (I \cap c, O \cap c, (P \cap c, T \cap c, A \cap (c \times c), M \cap c, F \cap c))$$

in effect by letting each member $c \in d_5(T, A)$ determine a process $x = (i, o, n) \in (d_6 X) d_5(T, A)$ whose alphabets and Petri net vertices are drawn from X but restricted to c .

If a process $X = (I, O, N)$ is implemented as an array of blocks each implementing a process $x = (i, o, n)$, then an alphabet ordering α on X determines a subset $(\mu (\alpha \uparrow I)^{-1} i)$ of input terminal indices associated with x and a subset $(\mu (\alpha \uparrow O)^{-1} o)$ of output terminal indices. Representing each block temporarily in the more awkward form of a triple

$$\langle ((\mu (\alpha \uparrow I)^{-1} i)^{\circ^{-1}}, ((\mu (\alpha \uparrow O)^{-1} o)^{\circ^{-1}}, \langle x \rangle) \rangle$$

of two lists of terminal indices and the unit list $\langle x \rangle$ in a set $q = d_7(\alpha, I, O) (d_6 X) d_5(T, A)$ given by

$$d_7 = \lambda(\alpha, I, O). \mu \lambda x. (\lambda(i, o, n). \langle ((\mu (\alpha \uparrow I)^{-1} i)^{\circ^{-1}}, ((\mu (\alpha \uparrow O)^{-1} o)^{\circ^{-1}}, \langle x \rangle) \rangle) x$$

fixes an ordered sequence $q^{\circ^{-1}}$ and hence a triple

$$t = b^* (q^{\circ^{-1}})^{\top} = b^* ((d_7(\alpha, I, O) (d_6 X) d_5(T, A))^{\circ^{-1}})^{\top}$$

such that $t_0, t_1 \in \mathbb{N}^*$ are input and output permutations respectively, and $t_2 \in \mathbb{D}^*$ is a list of processes extracted from X and ordered consistently with the permutations. This triple fully determines the decomposition $\mathcal{U}_m^\alpha X$ unless there are members of the alphabets I and O in the process $(I, O, N) = X$ that do not appear as transitions in its Petri net model N and hence not in the alphabets of any process $x \in (d_6 X) d_5(T, A)$.

Although a mismatch between the alphabets and the Petri net transitions is never desirable in a specification, an implementation that neglects the extra symbols would violate the specification and might allow design errors to escape notice. Whereas the alphabets of processes $x = (i, o, n) \in \mathcal{R}(t_2)$ are inferred entirely from N , we can correct for this effect by restoring the omitted symbols to the alphabets of the first term of t_2 by rewriting it to $d_8(I, O) t$ with

$$d_8 = \lambda(I, O). \lambda \langle p, r, (i, o, n) : s \rangle. \langle p, r, (i \cup (I - T), o \cup (O - T), n) : s \rangle$$

in a decomposition defined as follows.

$$\mathcal{U}_m^\alpha = \lambda X. (\lambda(I, O, (P, T, A, M, F)). d_8(I, O) b^* ((d_7(\alpha, I, O) (d_6 X) d_5(T, A))^{\circ^{-1}})^{\top}) X$$

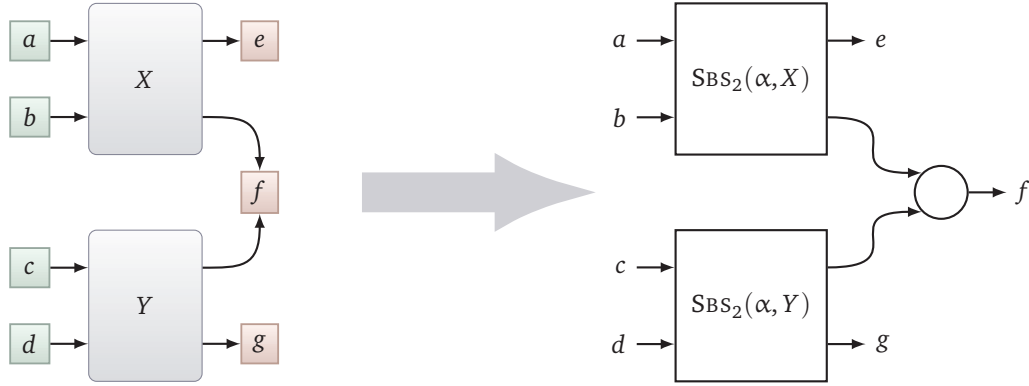


Figure 15.12: A process whose Petri net model consists of multiple disjoint components or of components joined only by way of open output transitions and dead-end places can be synthesized as separate blocks with synchronized outputs.

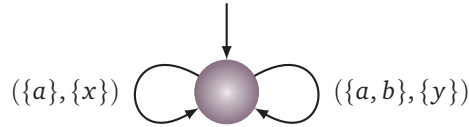


Figure 15.13: the simplest transducer with non-deterministic concurrency

15.6.2 Combining form

Because an alphabet ordering α for a process $X \in \mathbb{D}$ is necessarily also an alphabet ordering for each process $x \in \mathcal{R}((\mathcal{U}_m^\alpha X)_2)$, the latter could be implemented independently as $\text{SBS}_2(\alpha, x) \in \mathbb{H}$ by Equation 15.37 without further concern for its transducer level representations or other details, even when various optimizations are applicable. Hence it is feasible to bypass the construction of a combining form as such in favor of a generalized state based synthesis function $\text{SBS} : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$ defined as

$$\text{SBS} = \lambda(\alpha, X). (\lambda t. t_0^{-1} \times (\mathcal{F} \mathbf{R}) (\lambda x. \text{SBS}_2(\alpha, x))^* t_2 \times t_1^{-1}) \mathcal{U}_m^\alpha X \quad (15.38)$$

based on $t = \mathcal{U}_m^\alpha X$ consisting of two permutations and a list of processes x as explained above.

If state based synthesis were the only way of doing circuit synthesis, then the task to derive the transformation $\mathcal{F}_{\mathbb{D}\mathbb{H}}^\alpha : \mathbb{D} \rightarrow \mathbb{H}$ proposed at the beginning of this chapter could now be concluded by identifying it as

$$\mathcal{F}_{\mathbb{D}\mathbb{H}}^\alpha(X) = \text{SBS}(\alpha, X). \quad (15.39)$$

However, the next chapter explores further possibilities with a view to overcoming some limitations of state based synthesis, so it is prudent to be more tentative about this definition pending further investigation.

Statecraft

1. What is an example of a process $X \in \mathbb{D}$ whose implementation $SBS(\alpha, X)$ refines X under α but is not equivalent to it?
2. What would be the implications of cutting costs on the synthesis of a non-deterministic process by substituting a TOGGLE for each randomizer (Figure 13.14) in the controller block?
3. Suppose a user interacts with the transducer shown in Figure 15.13.
 - a) What acknowledgment should the user expect in response to inputs of a alone, b alone, or a and b concurrently?
 - b) Suppose concurrent inputs of a and b yield an acknowledgment of x . How should the user interact with the transducer subsequently?
 - c) Give a complete verbal account of the protocol, tabulate its anti-refined transducer model as in Table 15.1, and sketch the result as in Figure 15.5.
4. What is the triple (m, i, f) for each state of the intermediate representation of the feedback anti-refined transducer shown in Figure 15.5?
5. If two inputs to the circuit in Figure 15.6 are nearly simultaneous, the choice between concurrent and sequential acknowledgments is non-deterministic.
 - a) What is an example of an application for which a bias toward concurrency might benefit performance? (hint: [202])
 - b) The front end sequencer depends on an arbiter decomposition (Section 12.2) for its internal arbiter (Figure 13.12). What arbiter decomposition would favor concurrency and why? (hint: page 383)
6. Figure 15.12 shows a more general form of separable synthesis than the one derived in Section 15.6 in that it allows not just shared dead-end places between components but synchronized outputs.
 - a) How could the definition of \mathcal{U}_m^α be upgraded and a combining form Ω_m^α introduced to provide for this method of synthesis?
 - b) When would it be correct to combine common outputs by a MERGE instead of a JOIN, and how could the method be generalized further to cover this case? (hint: Figure 3.11)
 - c) A MERGE and a JOIN are both special cases of a completion detector. What would it take to cover all cases of outputs combined by unrestricted completion detectors?



I will not deny . . . that some Parts of
it might be contracted . . . But to
confess the Truth, I am now too lazy,
or too busy to make it shorter.

John Locke

CHAPTER

16

DIRECT MAPPING SYNTHESIS

To those readers disinclined to leave well enough alone and unfazed by further mathematical monkey business, there is still more to say about DI circuit synthesis. The method of direct mapping synthesis proposed in this chapter is relevant when a specification is computationally infeasible for state based synthesis as proposed in [Chapter 15](#) but is either too complicated or too unimportant to implement manually. The core idea of direct mapping synthesis is to transform a specification from a Petri net representation, typically generated by process combinators, directly to a netlist without using the reachability graph or transducer as intermediate representations. Although the resulting circuit may be larger, direct mapping synthesis can be computationally more efficient than state based synthesis because it avoids the burden of enumerating the process states entailed by these intermediate representations (the so called “state space explosion” problem [47]). Nestled inconspicuously in [Figure 3.1](#), this transformation deployed effectively leads to automated synthesis with unlimited scalability. On that note, this investigation concludes our treatment of the subject.

16.1 Overview

Direct mapping synthesis would seem to be a simple matter of inverting $\mathcal{F}_{\mathbb{L}\mathbb{D}}$, which transforms a netlist to a Petri net by [Equation 8.32](#), and indeed $\mathcal{F}_{\mathbb{B}\mathbb{L}} \circ \mathcal{F}_{\mathbb{B}}^\alpha$ expresses a transformation of this type by [Equation 8.20](#) and [Equation 9.8](#). The resulting netlist always has the form $\langle\langle I, O, B \rangle\rangle \in \mathbb{L}$ containing exactly one block $B \in \mathbb{B}$, but if B just happens to be refined by a primitive component ([Section 9.3.4](#)), then the job is done.

When beginners’ luck runs out, a different approach is needed. The second most obvious idea is to project the Petri net onto a whiteboard and look for patterns. By circling any primitive component Petri net shown in [Figure 9.7](#) or [Figure 9.8](#) appearing as a subgraph of the specification, we arrive at an implementation provided there is an exact cover (*cf.* [Figure 3.9](#)). The main drawback of this approach is that Petri nets generated by process combinators are not guaranteed to have an exact

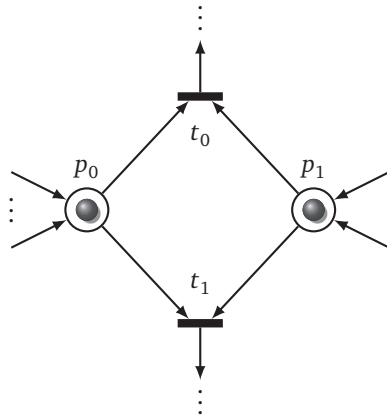


Figure 16.1: If p_0 grants a token transfer to t_0 and denies it to t_1 while p_1 grants it to t_1 and denies it to t_0 , both transitions have to rescind their requests and try another round.

cover by those of the primitives, and anecdotally the vast majority do not. Attempts to compensate by detecting more general patterns with known implementations and doing so automatically would depend on solving the subgraph isomorphism problem, a serious job in itself [152, 289]. Though maybe not impossible, this idea would probably limit scalability even if successful.

Closer to the right track, another idea for direct mapping synthesis is to have the circuit physically simulate the token flow through the Petri net with a block for each place and transition, a channel for each arc, and a protocol for all of them to follow. What protocol might that be? Places could notify their postset transitions when they hold a token, and a transition having received such a notification from all of its preset places could request the transfer of their tokens. A place subject to multiple concurrent requests for token transfers would need to arbitrate among them, and therefore would need to be able to deny a request. To avoid deadlock, a transition for which some but not all of its requests are denied would need to rescind the requests that have been granted (*cf.* “dining philosophers” [23, 112, 237]). A place having granted a rescinded request should send a new batch of notifications. The new notifications must not be sent to transitions that have not yet acknowledged their previous ones, because the previous notification could still be in transit on the wire. Who said anything about acknowledging the notifications? Maybe the protocol needs that condition too: a transition must acknowledge every notification even if it is not ready to request a transfer. For the same reason, the signal to rescind a request requires some acknowledgment from the place that granted it, and so on for every other exigency that comes to mind.

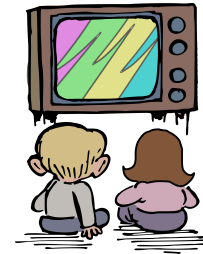


There is no need to ask whether all of the edge cases can be ironed out because this protocol already suffers from a fatal flaw. As Figure 16.1 shows, it could happen that two transitions with intersecting presets request token transfers concurrently, and if neither succeeds then both must try again. With only two transitions and two places, each round is like a coin toss, so the impasse is unlikely to persist. However, if there are ten transitions with ten places common to all of their

presets, then progress happens only in the one-in-a-billion chance of all ten places choosing to grant a request to the same transition.

One way forward would be to complicate the protocol by requiring the places in this situation to reach a consensus among themselves on the unique choice of a grant recipient. Distributed consensus algorithms in software have been well studied [29, 116, 153], but porting them to hardware would pose a formidable challenge. In the current setting, it would also require places to interact directly with other places and not just with transitions via the arcs. However, if other interactions than those mediated through the arcs are acceptable, then there is no need to take the trouble of implementing a distributed consensus algorithm because a simpler solution is possible.

The revised solution provides for any two or more transitions competing for the final token they need before firing to resolve the conflict through arbitration administered by a separate entity, which for the sake of this discussion is called a **monitor**. The monitor grants one transition's request, denies the other(s), and informs any other transition in receipt of the same token that it is no longer available. As explained in Section 16.6, the implementations of transition and monitor blocks observing this protocol are somewhat involved, but they have the advantage of allowing every arc to be implemented as just a wire and every place as not much more than a MERGE and a FORK. This solution is not ideally "distributed" insofar as multiple transitions may need to share a monitor, but a given Petri net typically can benefit from multiple independent monitors. If the transitions are partitioned into classes by transitive closure of preset intersection, then no transition competes with any from another class, so there can be a separate monitor for each class.



This solution permits a minor optimization. Each class of transitions related as above, their collective preset places, and the arcs connecting them can be designated as a **community** for lack of a better term. A large Petri net might have multiple communities of varying sizes. If any community is feasibly small, its share of the Petri net treated in isolation can be implemented by state based synthesis. If every community can be implemented by state based synthesis, the result generalizes the separation method described in Section 15.6. In any case, hereafter we may call them **state based synthetic communities** as opposed to **direct mapped synthetic communities**.

A limitation of this method is that it does not admit of a hard upper bound on spatial complexity of the synthesized circuit. The transition and monitor blocks are not of constant sizes but must be synthesized according to relationships specific to their communities, and their state spaces can vary considerably. In the extreme, even an individual monitor or transition could be too large for state based synthesis subject to constrained computational resources, requiring a recursive application of direct mapping synthesis as detailed in Section 16.7. Convergence in obstinate instances may demand a minimum fixed resource budget. Nevertheless, direct mapping synthesis offers a way to subdue state space explosion by an otherwise unavailable tradeoff between the complexity of the result and the cost of synthesizing it.

16.2 Mutual recurrences

Theoretically there is no need for any new math to express the algorithm for direct mapping synthesis sketched above, but we can make shorter work of it by a technique described in this section that is especially suited to process specifications given by systems of mutual recurrences as in Section 16.6.

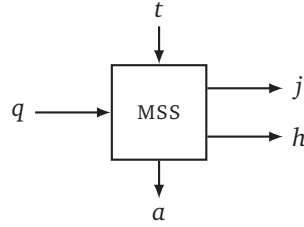


Figure 16.2: A mad scientist simulator can be in either Jekyll mode or Hyde mode. A toggle t changes the mode, a acknowledges a toggle, and j or h reports the mode in response to a query q .

For a running example, we have the mad scientist simulator shown in Figure 16.2. The scientist, known as Jekyll, is initially sane, but can be triggered by some event to become his insane alter-ego Hyde. A repetition of the same trigger restores his sanity, but he continues to change from Jekyll to Hyde or back again every time there is a trigger thereafter. The purpose of the simulator is to let the user ascertain the scientist’s state of mind without having to approach him. Applying a toggle input labeled t whenever the scientist experiences a trigger ensures that an output of either j or h in response to a query q always tracks the scientist’s current state of mind. As an extra feature, the simulator acknowledges every toggle input with the acknowledgment labeled a .

16.2.1 Ad hoc solution

This circuit may be simple enough to design manually, but as a warmup for others in this chapter that are not, let us proceed as if the only option is state based synthesis from a process style description. Decomposing the specification into small digestible sips, we might start by writing



$$S \equiv \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } j, S \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, M \rangle) \quad (16.1)$$

to define the simulator’s behavior when modeling the sanely disposed scientist. That is, the simulator accepts an input of q , responds with an output of j for Jekyll, and resumes acting like S , or accepts a toggle input t , acknowledges with a and then acts like a process M not written yet but soon to describe the scientist when mad. Although this definition technically is circular, the **fix** combinator should fix that shortly. First we finish up by writing the mad part.

$$M \equiv \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } h, M \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, S \rangle) \quad (16.2)$$

That is, when the scientist is mad, the simulator should accept an input of q , respond with h for Hyde, and keep acting like M , or should accept a toggle t , acknowledge with a , and start acting like S again. It is not completely straightforward to solve for S using the **fix** combinator as planned because the definition of M is also circular and also depends on S , but by a manipulation familiar to readers who have solved item 2 on page 120, we have

$$S = \text{fix } \lambda s. \text{alt} (\\ (\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } j, s \rangle, \\ (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, \text{fix } \lambda m. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } h, m \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, s \rangle) \rangle)$$

from which an implementation of the mad scientist simulator $\text{MSS} \in \mathbb{H}$ follows directly as

$$\text{MSS} = \text{SBS}(\langle t, q, j, h, a \rangle, S)$$

by Equation 15.38 with the arbitrarily selected alphabet ordering $\alpha = \langle t, q, j, h, a \rangle$ shown.

16.2.2 Solution by lists of functions

Whereas an *ad hoc* solution such as that of Section 16.2.1 is adequate for a pair of mutually dependent hand written recurrences, it is best to seek a generalized fixed point combinator for solving problems with large numbers of equations and unrestricted dependences among them. To say that a process $p \in \mathbb{D}$ satisfies a recurrence means there is some non-trivial function $e : \mathbb{D} \rightarrow \mathbb{D}$ for which $p \equiv e p$ holds. The analogous concept of a solution to a system of n mutual recurrences is captured by a list $P \in \mathbb{D}^n$ of n processes and a list $H \in (\mathbb{D}^n \rightarrow \mathbb{D})^n$ of n functions such that

$$P_i \equiv H_i P$$

holds for all $0 \leq i < n$. For the current example, the list $H \in (\mathbb{D}^2 \rightarrow \mathbb{D})^2$ consists of two functions

$$H = \langle \begin{array}{l} \lambda \langle s, m \rangle. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } j, s \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, m \rangle), \\ \lambda \langle s, m \rangle. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } h, m \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, s \rangle) \end{array} \rangle$$

each taking two processes, and the solution $P = \langle S, M \rangle \in \mathbb{D}^2$ is the list of two processes S and M satisfying $S \equiv H_0 \langle S, M \rangle$ and $M \equiv H_1 \langle S, M \rangle$ with S as obtained above and M similar. Just as the usual fixed point combinator yields the solution $p = \text{fix } e$ in the simple case, the solution $P = F H$ in the general case should be given by a fixed point combinator $F : (\mathbb{D}^n \rightarrow \mathbb{D})^n \rightarrow \mathbb{D}^n$. However, we are usually interested in only one process from among those constituting the solution to the system, such as S but not M in the current example, and it simplifies the derivation to seek a fixed point combinator limited to

$$\dot{\Upsilon} : (\mathbb{D}^* \rightarrow \mathbb{D})^* \rightarrow \mathbb{D}$$

taking a list of functions H and yielding only the first process $P_0 = \dot{\Upsilon} H$ from the list called P above. The generalized fixed point combinator is denoted by the Greek letter upsilon with a dot over it because a better alternative without the dot is coming up in Section 16.2.3.

To derive the fixed point combinator $\dot{\Upsilon}$, we may reason as follows. If H were a list of only one function H_0 , then H_0 would take a list $\langle p \rangle$ of only one process as an argument, and we could use the usual fixed point combinator to obtain the solution

$$P_0 = \dot{\Upsilon}(H) = \text{fix } \lambda p. H_0 \langle p \rangle.$$

If H were to contain some number $n > 1$ of functions but somehow we already knew the rest of the processes $P_1, P_2 \dots P_{n-1}$ satisfying $P_i \equiv H_i P$, we could treat them as constants and still use the usual fixed point combinator to obtain P_0 as the fixed point of a function parameterized by a single unknown process p .

$$P_0 = \text{fix } \lambda p. H_0 \langle p, P_1, P_2 \dots P_{n-1} \rangle \tag{16.3}$$

In the mad scientist simulator example, this solution would be like writing

$$S = \text{fix } \lambda p. (\lambda \langle s, m \rangle. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } j, s \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, m \rangle)) \langle p, M \rangle$$

as if some constant process M representing Hyde mode were already known. If P_0 and P_1 were both unknown but $P_2 \dots P_{n-1}$ were known and $\dot{\Upsilon}$ were computable for lists of fewer than n functions, we could solve [Equation 16.3](#) for P_0 by eliminating P_1 from it with the substitution

$$P_1 = \dot{\Upsilon} (\lambda h. h \circ \lambda q. p : q)^* (H \ll 1)$$

which can be evaluated with $\dot{\Upsilon}$ because each of the $n - 1$ functions $h \in \mathcal{R}(H \ll 1)$ determines a function $h \circ \lambda q. p : q$ taking a list q of length $n - 1$ as an argument.

A generalization of this substitution can be used to eliminate each of P_2 through P_{n-1} from [Equation 16.3](#), making $\dot{\Upsilon}$ expressible as a recurrence. We note first that the i -th fixed point determined by a list $x \in (\mathbb{D}^{|x|} \rightarrow \mathbb{D})^*$ of functions is given by the application of $\dot{\Upsilon}$ to a list $r^i x$ derived from x by rolling it left i times, with

$$r = \lambda l. (l \ll 1) \parallel \langle l_0 \rangle$$

temporarily denoting the function that rolls a list once to the left, and having each term $h \in \mathcal{R}(x)$ compensate by rolling its argument i times to the right before operating on it.

$$\dot{\Upsilon} (\lambda h. h \circ r^{|x|-i})^* r^i x$$

Specifically with respect to [Equation 16.3](#), we would substitute

$$P_i = \dot{\Upsilon} (\lambda h. h \circ r^{|H|-i})^* r^{i-1} ((\lambda h. h \circ (\lambda q. p : q))^* (H \ll 1))$$

with i ranging from 1 to $n - 1$, or more succinctly

$$P_i = \dot{\Upsilon} r^{i-1} (\lambda h. h \circ (\lambda q. p : q) \circ r^{|H|-i})^* (H \ll 1).$$

These observations suggest the recursive definition for the fixed point combinator

$$\dot{\Upsilon}(H) = (\lambda r. \text{fix } \lambda p. H_0(p : (\lambda i. \dot{\Upsilon} r^{i-1} (\lambda h. h \circ (\lambda q. p : q) \circ r^{|H|-i})^* (H \ll 1)) \iota_{|H \ll 1|}^1)) \lambda l. (l \ll 1) \parallel \langle l_0 \rangle$$

whereby a precise expression of the solution to our running example would be

$$S = \dot{\Upsilon} \langle \lambda \langle s, m \rangle. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } j, s \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, m \rangle), \lambda \langle s, m \rangle. \text{alt} ((\mathcal{F} \text{ seq}) \langle \text{get } q, \text{put } h, m \rangle, (\mathcal{F} \text{ seq}) \langle \text{get } t, \text{put } a, s \rangle) \rangle.$$

16.2.3 Solution by dependence graphs

The generalized fixed point combinator derived in [Section 16.2.2](#) implies an algorithmic solution to any system of recurrences that relieves some of the manual effort, but still requires explicitly transcribing each equation in the system. Under certain reasonable conditions, we can specify the system in more flexible and intuitive terms without needless repetition.

Dependence graphs

A notable feature of [Equation 16.1](#) and [Equation 16.2](#) is that both equations have the form of a choice between sequential compositions, each sequential composition concludes with a “mode”, and each equation is said to depend on the other. Taking these notions literally evokes the image

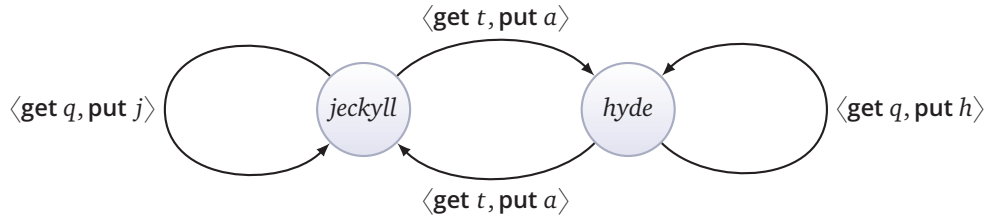


Figure 16.3: A dependence graph with edges labeled by lists of processes and vertices of any type represents a system of recurrences.

of a dependence graph as shown in Figure 16.3, with edges labeled by lists of processes meant to execute in sequence *en route* between vertices of some abstract type representing the modes. If this graph is formalized as a member g of a set

$$\mathcal{P}(V \times \mathcal{P}(\mathbb{D}^* \times V))$$

following the usual adjacency set convention, we can almost envision transforming it automatically to a system of recurrences solvable by Υ . Moreover, the graph representation might be easier to construct because it would enable a choice of vertices with more meaningful interpretations than mere indices into a list. For the current running example, we could choose a set $V = \{jeckyll, hyde\}$ without demanding any concrete model for it, but more generally a vertex could be structured to keep track of any relevant semantic information.

Restrictions on dependence graphs

Two conditions make this idea workable, neither of which impedes its subsequent use in this chapter. One condition is that the set V of vertices must be totally ordered with the minimum value conforming to the desired solution. Stipulating $jeckyll < hyde$ by definition would suffice for the example. The other is a technical condition enabling any adjacency set to determine a list: the processes used in the edge labels must also be totally ordered, even though \mathbb{D} generally is not. This condition is satisfied by a restriction to the set

$$\mathbb{D} \cap (\mathcal{P}(\mathbb{G}) \times \mathcal{P}(\mathbb{G}) \times \mathbb{P})$$

of processes in \mathbb{D} whose alphabets contain only generic symbols (Section 8.7.1) and hence a graph restricted to $g \in \mathcal{P}(V \times \mathcal{P}((\mathbb{D} \cap (\mathcal{P}(\mathbb{G}) \times \mathcal{P}(\mathbb{G}) \times \mathbb{P}))^* \times V))$, for example

$$g = \{ (jeckyll, \{ \langle \langle \text{get } q, \text{put } j \rangle, jeckyll \rangle, \langle \langle \text{get } t, \text{put } a \rangle, hyde \rangle \}), (hyde, \{ \langle \langle \text{get } q, \text{put } h \rangle, hyde \rangle, \langle \langle \text{get } t, \text{put } a \rangle, jeckyll \rangle \}) \}$$

where q, j, t, a , and h are chosen to be members of \mathbb{G} .

Solution

The first step to solving a system of recurrences expressed as a graph g is to transform it to a list $H : (\mathbb{D}^n \rightarrow \mathbb{D})^n$ of $n = |g|$ functions solvable by Υ . Each vertex $(m, e) \in g$ with adjacency set e

induces a term $H_i : \mathbb{D}^n \rightarrow \mathbb{D}$ with index $i = \mathcal{D}(g)^\circ m$ that should be something of the form

$$H_i = \lambda P. (\mathcal{F} \text{ alt}) \langle (\mathcal{F} \text{ seq}) (\dots), \dots \rangle \quad (16.4)$$

in terms of a formal parameter $P \in \mathbb{D}^n$ with details dependent on $e = (\Psi g) m$. In particular, each edge $(p, q) \in e$ labeled by a list $p \in \mathbb{D}^*$ of processes and terminating at a vertex $q \in \mathcal{D}(g)$ accounts for a term

$$(\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)$$

in Equation 16.4 with the index $j = \mathcal{D}(g)^\circ q$ referring to the process in the formal parameter P that corresponds to the lexicographically j -th vertex in the graph, which is the terminus q . Hence the full set of sequential composition terms would be

$$\bigcup_{(p,q) \in (\Psi g) m} \{(\lambda j. (\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)) \mathcal{D}(g)^\circ q\}$$

from which the list of processes appearing in Equation 16.4 follows as

$$\left(\bigcup_{(p,q) \in (\Psi g) m} \{(\lambda j. (\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)) \mathcal{D}(g)^\circ q\} \right)^{\circ-1}$$

and hence the whole function H_i as

$$H_i = \lambda P. (\mathcal{F} \text{ alt}) \left(\bigcup_{(p,q) \in (\Psi g) m} \{(\lambda j. (\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)) \mathcal{D}(g)^\circ q\} \right)^{\circ-1}$$

or the whole list of functions as

$$H = \left(\lambda m. \lambda P. (\mathcal{F} \text{ alt}) \left(\bigcup_{(p,q) \in (\Psi g) m} \{(\lambda j. (\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)) \mathcal{D}(g)^\circ q\} \right)^{\circ-1} \right)^* \mathcal{D}(g)^{\circ-1}.$$

The only other step is to solve the system of recurrences H for a fixed point. It is worthwhile to encapsulate both steps by an alternative generalized fixed point combinator Υ defined as follows.

$$\Upsilon(g) = \dot{\Upsilon} \left(\lambda m. \lambda P. (\mathcal{F} \text{ alt}) \left(\bigcup_{(p,q) \in (\Psi g) m} \{(\lambda j. (\mathcal{F} \text{ seq}) (p \parallel \langle P_j \rangle)) \mathcal{D}(g)^\circ q\} \right)^{\circ-1} \right)^* \mathcal{D}(g)^{\circ-1} \quad (16.5)$$

16.3 Refined canonical forms

Before proceeding with more of the details of direct mapping synthesis, we need to be explicit about a couple of conditions. The first condition is that the method to be proposed requires an open Petri net modeled DI process

$$X \in \mathbb{D} \cap (\mathcal{P}(\mathbb{T}) \times \mathcal{P}(\mathbb{T}) \times \hat{\mathbb{P}})$$

by Equation 5.7, meaning that input transitions have empty presets and output transitions have empty postsets. This set includes but is not limited to $\tilde{\mathbb{D}}$ by Equation 5.8 because it does not require Petri nets in $\tilde{\mathbb{P}}$ with mutually disjoint presets or postsets among observable transitions as specified in Equation 5.9. By way of a rationale, each input transition is to be implemented by a block with a single input terminal and each output transition by a block with a single output terminal, all



Figure 16.4: Deleting terminal places refines the specification.

exposed to the environment. An input transition controlled by the environment can not be enabled or inhibited from firing by anything within the circuit, so there would be nothing for its preset places to do if it had any. Whereas a closed Petri net constrains the specified environment, an implementation can do no such thing to the actual environment, so it would be meaningless to attempt direct mapping in this style starting from a closed Petri net specification.

This condition turns out to have certain implications. Theoretically the requirement for an open Petri net model imposes no restriction on the class of specifications that can be implemented, because any member of \mathbb{D} can be converted to a behaviorally equivalent member of $\tilde{\mathbb{D}}$ by Equation 7.36. In practice this conversion is never appropriate for direct mapping synthesis because it requires enumerating the state space. If enumerating the state space were feasible, then state based synthesis would be the better option. If the specification is given by a closed Petri net whose state space enumeration is infeasible, then the outlook is indeed grim, but this eventuality is always preventable for specifications expressed in terms of process combinators by banishing the `env` combinator from the expression, which guarantees a result in $\tilde{\mathbb{D}}$. For example, instead of writing Equation 9.16 for a 2-by-1 decision wait specification, we could write

$$X = \text{loop alt} (\text{seq} (\text{par} (\text{get } r_0, \text{get } c_0), \text{put } d_{00}), \text{seq} (\text{par} (\text{get } r_1, \text{get } c_0), \text{put } d_{10}))$$

and ignore the environment. The result after local optimization is shown at the right of Figure 16.4, with Equation 9.16 transformed to an open Petri net and optimized for convenient comparison at the left. As this example illustrates, omitting the environment is easy and results in a refinement but still meets the specification. The downside is that the implementation may be more costly than necessary. Unlike a decision wait, the refinement in this example needs an arbiter in it somewhere because it arbitrates between concurrent row inputs. This cost is the price of avoiding state space enumeration.

This example raises the question of whether it would be possible to avoid this cost by specifying a process like the one on the left of Figure 16.4 as

$$X = \text{loop} (\mathcal{F} \text{alt}) \text{seq}^* \langle \begin{array}{l} (\text{par} (\text{get } r_0, \text{get } c_0), \text{put } d_{00}), \\ (\text{par} (\text{get } r_1, \text{get } c_0), \text{put } d_{10}), \\ (\text{par} (\text{get } r_0, \text{get } r_1), \perp) \end{array} \rangle \quad (16.6)$$

for a suitably defined divergent process \perp , which yields an open Petri net with no need for a conversion involving state space enumeration from a closed Petri net. Although the marked place and the anonymous transition express the constraint that the row inputs must not be concurrent, physically emulating them in hardware accomplishes nothing. If the transition were ever to fire, then the environment would have already violated the specification, so the transition might as well not be there.

This observation leads to the second condition about process specifications required for the proposed method of direct mapping synthesis: there can be no terminal places in the Petri net model. Fortunately, transforming a Petri net to meet this condition is no more difficult than any of the Petri net optimizations described in [Section 9.1](#), although technically not an optimization because it yields a refinement rather than a behaviorally equivalent result. For a Petri net $N = (P, T, A, M, F) \in \mathbb{P}$, the terminal places are precisely the set $r = P - \mathcal{D}(A)$, and are removed by rewriting N to

$$N \dot{-} (r, \emptyset, \emptyset, \emptyset, \emptyset)$$

by [Equation 5.18](#). However, removing a terminal place by itself could leave open an anonymous transition, contrary to the assumptions of well formed Petri nets stipulated in [Section 5.2.1](#). Rewriting N further to

$$N \dot{-} (\emptyset, r', \emptyset, \emptyset, \emptyset)$$

with $r' = (T \cap \mathbb{V}) - \mathcal{D}(A)$ would remove this transition as well, but might expose more terminal places formerly belonging to its preset. To ensure a clean sweep, we should iterate both steps exhaustively by rewriting N to $\chi_8 N$ with

$$\chi_8 = (\lambda N. N \dot{-} (\lambda r. (r, r, \emptyset, \emptyset, \emptyset))) (\lambda (P, T, A, M, F). (P \cup T \cap \mathbb{V}) - \mathcal{D}(A)) N^\infty$$

denoting one last Petri net transformation to follow those given in [Section 9.1](#). An unrestricted process $X = (I, O, N) \in \mathbb{D}$ with its Petri net model $N \in \mathbb{P}$ refined accordingly and optimized by [Equation 9.7](#) for good measure is denoted $\mathbf{RP}(X)$ hereafter with $\mathbf{RP} : \mathbb{D} \rightarrow \mathbb{D}$ given by

$$\mathbf{RP}(X) = (\lambda (I, O, N). (\lambda (i, o, n). (i, o, \chi_8 \chi_{\mathbb{P}} n))) (\lambda k. \langle \mathbf{P}X, X \rangle_{\delta_k^\emptyset}) (\{N\} - \hat{\mathbb{P}}) X \quad (16.7)$$

and is called the **refined canonical form** of X , although the result does not depend on the canonical form $\mathbf{P}X$ given by [Equation 7.36](#) unless the Petri net N is closed.

16.4 Decomposition

Following the algorithm sketched in [Section 16.1](#), the first step toward direct mapping synthesis is to find the partition of greatest cardinality on the transitions in a Petri net whereby any two transitions whose presets intersect belong to the same class. For a Petri net $N = (P, T, A, M, F) \in \mathbb{P}$ with arcs A appearing in a candidate process $\mathbf{RP}(X) = (I, O, N) \in \mathbb{D}$ to be synthesized, a function

$$p = \Psi \Pi (\mu \lambda (i, j). (j, i)) A \quad (16.8)$$

takes any place or transition $v \in P \cup T$ to its preset $\bullet v = p v$ by [Equation 6.1](#) and [Equation 6.7](#). It will be necessary to list the classes lexicographically according to an alphabet ordering $\alpha \in \mathbb{T}^*$ presumably specified along with the process to be synthesized, which is convenient to extend to a list $a \in (\mathbb{T} \cup \mathbb{V})^*$ given by

$$a = \flat \langle \alpha \uparrow T \cap \mathcal{D}(A), \alpha \uparrow T - \mathcal{D}(A), (T - \mathcal{R}(\alpha))^{\circ-1} \rangle \quad (16.9)$$

in terms of the transitions T to include in its range any unobservable transitions in T following the inputs and outputs. Hereafter an auxiliary decomposition function

$$\dot{U}_z^\alpha = \lambda(T, A). (\langle \alpha \uparrow T \cap \mathcal{D}(A), \alpha \uparrow T - \mathcal{D}(A), (T - \mathcal{R}(\alpha))^{\circ-1} \rangle, \Psi \Pi (\mu \lambda(i, j). (j, i)) A) \quad (16.10)$$

is used to denote the pair (a, p) of these two functions by $\dot{U}_z^\alpha(T, A)$ in terms of the transitions and the arcs of the relevant Petri net.

Building the partition is now straightforward. Any transition $t \in T$ whose preset $p t$ by [Equation 16.8](#) contains a place $i \in p t$ necessarily belongs to the same class as every member of the *postset* of i given by $(\Psi \Pi A) i$. Any member $j \in T$ of a class in the partition therefore determines the whole class by the percolation

$$(\rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \{j\}$$

by [Equation 6.4](#) with the whole partition following as

$$(\mu \rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \bigcup_{j \in T} \{\{j\}\} \in \mathcal{P}(\mathcal{P}(\mathbb{T} \cup \mathbb{V})).$$

To make a list of the classes, we map each transition temporarily to its ordinal with respect to a by [Equation 16.9](#)

$$(\mu^2 a^{-1}) (\mu \rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \bigcup_{j \in T} \{\{j\}\} \in \mathcal{P}(\mathcal{P}(\mathbb{N}))$$

so that the classes can be listed lexicographically

$$((\mu^2 a^{-1}) (\mu \rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \bigcup_{j \in T} \{\{j\}\})^{\circ-1} \in \mathcal{P}(\mathbb{N})^*$$

and then converted back to a list of sets of transitions

$$(\mu a)^* ((\mu^2 a^{-1}) (\mu \rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \bigcup_{j \in T} \{\{j\}\})^{\circ-1} \in \mathcal{P}(\mathbb{T} \cup \mathbb{V})^*$$

which we identify as the decomposition $\dot{U}_z^\alpha(T, A)$ according to the decomposition function

$$\dot{U}_z^\alpha = \lambda(T, A). \left(\lambda(a, p). (\mu a)^* ((\mu^2 a^{-1}) (\mu \rho \lambda t. \bigcup_{i \in p t} (\Psi \Pi A) i) \bigcup_{j \in T} \{\{j\}\})^{\circ-1} \right) \dot{U}_z^\alpha(T, A). \quad (16.11)$$

16.5 Interacting state based synthetic communities

If an optimistic method of direct mapping synthesis described in this section succeeds in practice, there may be no need to look further. Under favorable conditions, the decomposition $\dot{U}_z^\alpha(T, A)$ takes a large Petri net to a large number of very small classes of transitions, so small that each class by itself induces a feasible candidate for state based synthesis. Then direct



mapping synthesis reduces to combining a list $y \in \mathbb{H}^*$ of state based synthetic blocks into a block $\Omega_z^\alpha((T,A), y) \in \mathbb{H}$ by a combining form

$$\Omega_z^\alpha : (\mathcal{P}(\mathbb{T} \cup \mathbb{V}) \times \mathcal{P}((\mathbb{T} \cup \mathbb{V}) \times (\mathbb{T} \cup \mathbb{V}))) \times \mathbb{H}^* \rightarrow \mathbb{H}$$

derived in [Section 16.5.4](#). Before that, [Section 16.5.1](#) discusses the front end of each block, corresponding roughly to the places in each community, [Section 16.5.2](#) describes the back end pertaining to the transitions, and [Section 16.5.3](#) briefly puts them together. However, if the assumption of small classes proves overly optimistic, the combining form and the front ends are of further use in [Section 16.6](#).

16.5.1 Places

Given the plan for each arc in a Petri net to be implemented as a wire and each transition to broadcast a signal on the wires implementing its outgoing arcs when it fires, we must envision each place having a MERGE network to gather the signals from the transitions in its preset. A MERGE network is appropriate because a place changes from unmarked to marked whenever any one transition in its preset fires. A MERGE network may diverge if multiple transitions in the preset fire concurrently, but then so too does the process we intend to implement because the environment has caused a safety violation to its Petri net model, leaving the implementation under no further obligation.

Whereas the MERGE network clearly accounts for the front end of the block implementing a place, the back end may be less intuitive. If a place simulates transferring its token by sending a signal, then it should send the signal through only one of its outgoing arcs because only one transition in its postset can fire. However, as noted in [Section 16.1](#), the attempt to develop a protocol along these lines ends badly. If instead we envision a place transmitting a signal only to announce the availability of its token, then it makes sense for the place to broadcast the signal to every transition in its postset by way of a FORK network. This protocol presupposes the postset transitions electing the token beneficiary among themselves, with no need to inform the place of the outcome.

Deferring consideration of the back end FORK network for the moment, we might write

$$\text{MERGE } |p \ i|$$

to express the front end MERGE network of a place i with preset $p \ i$ and p defined by [Equation 16.8](#) relative to the Petri net being implemented were it not for a couple of edge cases. Although [Equation 16.7](#) rules out places with empty postsets, there could still be places i with presets $p \ i = \emptyset$. A MERGE with no inputs is undefined, but an alternative block

$$\text{Z FORK}$$

with no input terminals and an output that never transmits a signal serves a similar purpose, suggesting the more general expression

$$\langle \text{MERGE } |p \ i|, \text{Z FORK} \rangle_{\delta_\emptyset^{p \ i}}.$$

Usually a place with an empty preset would be written out of the Petri net by dead code elimination ([Section 9.1.7](#)) unless it is initially marked. An initially marked place $i \in M$ indicated as such by its membership in the set M of initially marked places associated with the Petri net can be

made to transmit an initial notification of its token if it is put in series with a PUSH. An expression encompassing marked or unmarked places with empty or non-empty presets would be

$$F\langle\langle\text{MERGE } |p \ i|, \text{Z FORK}\rangle_{\delta_{\emptyset}^{pi}}, \langle\text{I, PUSH}\rangle_{\delta_{\emptyset}^{(i)-M}}\rangle$$

and an expression for a list of blocks ordered lexicographically by the places they represent with one for each place in the Petri net would be

$$(\lambda i. F\langle\langle\text{MERGE } |p \ i|, \text{Z FORK}\rangle_{\delta_{\emptyset}^{pi}}, \langle\text{I, PUSH}\rangle_{\delta_{\emptyset}^{(i)-M}}\rangle)^* v^{\circ-1}$$

in terms of the set $v = \bigcup \mathcal{R}(p^* a)$ of places inferred from the alphabet ordering a defined by Equation 16.9.

To restrict the places to those associated with a particular class of transitions $u \in \mathcal{U}_z^\alpha(T, A)$, we need only use values of

$$(a, p) = \check{\mathcal{U}}_z^\alpha(u, A) \quad (16.12)$$

local to u in these expressions instead of those that pertain to the whole Petri net, but this restriction leads to another edge case. If a class u contains an input transition, then the input transition has an empty preset according to Equation 16.7, which can intersect no other presets, so there can be no other transitions in the class by Equation 16.11 and the union v of presets over the class can only be empty. Whereas we might normally write

$$(\mathcal{F} \mathbf{R}) (\lambda i. F\langle\langle\text{MERGE } |p \ i|, \text{Z FORK}\rangle_{\delta_{\emptyset}^{pi}}, \langle\text{I, PUSH}\rangle_{\delta_{\emptyset}^{(i)-M}}\rangle)^* v^{\circ-1}$$

for the combined array of blocks implementing the front ends of the places, this expression is undefined for a vanishing v . In anticipation of the array being cascaded with something that implements the transition, and of there being only one input terminal on whatever that is, a wire can stand in for the the array in that case and then be denoted $\text{PL}(M, a, p)$ in terms of the following function.

$$\text{PL} = \lambda(M, a, p). (\lambda v. \langle(\mathcal{F} \mathbf{R}) (\lambda i. F\langle\langle\text{MERGE } |p \ i|, \text{Z FORK}\rangle_{\delta_{\emptyset}^{pi}}, \langle\text{I, PUSH}\rangle_{\delta_{\emptyset}^{(i)-M}}\rangle)^* v^{\circ-1}, \text{I}\rangle_{\delta_{\emptyset}^i}) \bigcup \mathcal{R}(p^* a) \quad (16.13)$$

16.5.2 State based transition arrays

The next step to implementing a process X by state based synthetic communities is to build a block $\text{TRA}_0(v, a, p) \in \mathbb{H}$ (mnemonic for “transition array”) implementing by state based synthesis the transitions associated with a class u , an alphabet ordering a , and a preset function p as above, and with

$$v = \bigcup \mathcal{R}(p^* a) \quad (16.14)$$

containing all places in the presets of members of u , so that an array of blocks each of the form

$$\mathbf{C}_{|v|+\delta^v} \langle \text{PL}(M, a, p), \text{TRA}_0(v, a, p) \rangle$$

implementing a single community can be combined to implement the process X as developed shortly in Section 16.5.4. The mnemonic is subscripted TRA_0 to indicate a restriction of the more general alternative developed in Section 16.6.2 to arrays of transitions whose state based synthesis is computationally feasible. Here again we must make explicit provision for the case of a class u whose

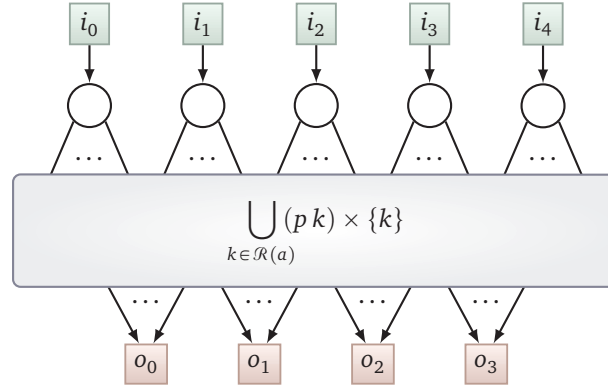


Figure 16.5: Arcs from places v to transitions $\mathcal{R}(a)$ isolated from a larger Petri net temporarily determine a localized Petri net for state based synthesis with fabricated alphabets i and o .

presets v are empty because it contains only an open input transition. The circuit corresponding to the set of a single transition reduces to a wire connected by $\delta_{\emptyset}^v = 1$ line to the array of places, which also reduces to a wire as noted above.

Drawing a boundary in effect separating the places on the front end from the transitions on the back end, this organization in the non-degenerate case narrows the responsibility of the state based synthesis algorithm to a block with exactly one input terminal for each place and one output for each transition as shown in Figure 16.5. The MERGE networks and PUSH primitives needed for places with multiple predecessors or with initial markings are delegated to the block $PL(M, a, p)$.

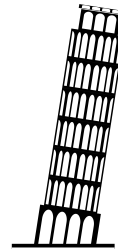
To synthesize the back end not from a process in \mathbb{D} nor even from a Petri net but from only a fragment of a Petri net, we need something like a wrapper around the fragment. Intuitively the wrapper would be a process whose Petri net model has transitions $u = \mathcal{R}(a)$, places v by Equation 16.14 and arcs

$$\bigcup_{k \in \mathcal{R}(a)} (p k) \times \{k\}$$

copied from the original Petri net model of X , which would fix the output alphabet of the process as $\mathcal{R}(a)$ as well. The input transitions (labeled i in Figure 16.5) might not map bijectively to any particular set of transitions from the original, but could be an improvised list of $|v|$ generic terminals such as $\mathbb{G}^{\circ^{-1} *}_{l_{|v|}}$.

A few technicalities prevent the construction from being quite so simple. The transitions in a are not necessarily observable in X even though the wrapper needs an observable output transition for each term of a . Hence we have to substitute generic terminals for the terms of a and rewrite the arcs accordingly. Some transitions in a might be observable with no need for substitution, and some of those may be generic, but it simplifies matters to avoid clashes with them in the improvised alphabets by defining a pair

$$(i, o) = (f_0 v) a \in \mathbb{G}^{|v|} \times \mathbb{G}^{|a|}$$



with all terms of a substituted regardless according to

$$f_0 = \lambda v. \lambda a. (\lambda s. (s \uparrow |v|, s \ll |v|)) (\lambda k. \mathbb{G}^{\circ-1*} t_{|v|+|a|}^{k+1}) \max(\{0\} \cup (\mu \mathbb{G}^{\circ}) (\mathbb{G} \cap \mathcal{R}(a)))$$

so that the wrapper process can have input and output alphabets $\mathcal{R}(i)$ and $\mathcal{R}(o)$ respectively, and can be synthesized with an alphabet ordering $i \parallel o$. In addition to the arcs noted above, there needs to be an incident arc on each place $j \in v$ from the $(v^\circ j)$ -th term of i as shown in [Figure 16.5](#). Putting both sets of arcs together, we would have

$$\bigcup_{j \in v} \{(i v^\circ j, j)\} \cup \bigcup_{k \in \mathcal{R}(a)} (pk) \times \{k\}$$

were it not for the need to rewrite those terminating on transitions $l \in \mathcal{R}(a)$ to the corresponding generic terminal $o a^{-1} l$. Incorporating these arcs into a Petri net with places v and transitions $\mathcal{R}(i \parallel a)$ permits an overall rewrite to

$$(\lambda l. \langle l, o a^{-1} l \rangle_{\delta_{\emptyset}^{\{l\}-\mathcal{R}(a)}})^{\bullet} (v, \mathcal{R}(i \parallel a), \bigcup_{j \in v} \{(i v^\circ j, j)\} \cup \bigcup_{k \in \mathcal{R}(a)} (pk) \times \{k\}, \emptyset, \emptyset) \in \mathbb{P}$$

by [Equation 5.10](#), which can be abbreviated as $((f_1 p) a) (i, o, v)$ according to

$$f_1 = \lambda p. \lambda a. \lambda (i, o, v). (\lambda l. \langle l, o a^{-1} l \rangle_{\delta_{\emptyset}^{\{l\}-\mathcal{R}(a)}})^{\bullet} (v, \mathcal{R}(i \parallel a), \bigcup_{j \in v} \{(i v^\circ j, j)\} \cup \bigcup_{k \in \mathcal{R}(a)} (pk) \times \{k\}, \emptyset, \emptyset) \quad (16.15)$$

enabling a state based synthetic block (by [Equation 12.3](#))

$$(\lambda \langle (i, o), n \rangle. \text{SBS} (i \parallel o, (\mathcal{R}(i), \mathcal{R}(o), n(i, o, v)))) (\langle f_0 v, f_1 p \rangle^{\Delta} a^{\Delta}) \in \mathbb{H}$$

to meet the requirements for the back end when v is non-empty. When v is empty, the input alphabet i as given by $(f_0 v) a$ is empty, the Petri net $((f_1 p) a) (i, o, v)$ is not well formed, and the block is undefined unless we generalize it to $\text{TRA}_0(v, a, p)$ by

$$\text{TRA}_0 = \lambda (v, a, p). \langle \lambda \langle (i, o), n \rangle. \text{SBS} (i \parallel o, (\mathcal{R}(i), \mathcal{R}(o), n(i, o, v)))) (\langle f_0 v, f_1 p \rangle^{\Delta} a^{\Delta}), \mathbf{1} \rangle_{\delta_{\emptyset}} \quad (16.16)$$

for reasons noted above.

16.5.3 Communities

With these minor issues resolved, an implementation for a community including the front and back ends by state based synthesis and covering the degenerate case follows as a block

$$\text{COM}_0(\alpha, A, M) u \in \mathbb{H}$$

in terms of an alphabet ordering α and a class $u \in \mathcal{U}_z^{\alpha}(T, A)$ of transitions from a Petri net with transitions T , arcs A , and initial marking M based on $\text{COM}_0(\alpha, A, M) : \mathcal{P}(\mathbb{T} \cup \mathbb{V}) \rightarrow \mathbb{H}$ defined as

$$\text{COM}_0(\alpha, A, M) = \lambda u. (\lambda (a, p). (\lambda v. \mathbf{C}_{|v|+\delta_{\emptyset}^v} \langle \text{PL}(M, a, p), \text{TRA}_0(v, a, p) \rangle \rangle \bigcup \mathcal{R}(p^* a)) \dot{\mathcal{U}}_z^{\alpha}(u, A).$$

When state based synthesis is feasible, a block $\text{COM}_0(\alpha, A, M) u$, mnemonic for “community”, can serve as a plug-compatible replacement for the more general alternative $\text{COM}(\alpha, A, M) u$ to be developed in [Section 16.6.6](#).

16.5.4 Combining form

A combining form capturing the notion of assembling an arbitrary Petri net from its constituent communities is a tall order due to the lack of any repetitive or recurrent structure in general. Starting from an array of blocks with each one corresponding to a community, we proceed in this section over several steps involving permutation networks, feedback paths and other miscellany to derive a direct mapping synthesis function $\text{DMS}_0 : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$ by the end.

Community array

The step from describing an individual community to describing the ensemble of interacting communities implementing a specification $X \in \mathbb{D}$ modeled as above is largely achieved by making a list

$$y = \text{COM}_0(\alpha, A, M)^* c \in \mathbb{H}^*$$

of blocks determined by a decomposition

$$c = \mathcal{U}_z^\alpha(T, A) \in \mathcal{P}(\mathbb{T} \cup \mathbb{V})^* \quad (16.17)$$

into an array ($\mathcal{F} \mathbf{R}$) $y \in \mathbb{H}$. This array would have one output terminal for each transition in the Petri net, with the first $|c_0|$ output terminals being those of the transitions in c_0 , the next $|c_1|$ in c_1 , and so on.

Output permutation network

The first step toward transforming this array into an implementation of the process X is to find an output permutation network rearranging these out-of-order terminals consistently with the alphabet ordering a by [Equation 16.9](#). A list

$$(\mu a^{-1})^* c \in \mathcal{P}(\mathbb{N})^{|c|}$$

of sets of numbers expresses the alphabet ordinals of each class of transitions in order of the class's position in c , and a flatter list

$$\overset{\circ}{b}(\mu a^{-1})^* c \in \mathbb{N}^{|a|}$$

gives the ordinals of the transitions in the order they would appear on the back of the array without the benefit of any output permutation network.

Because not all of the terminals on the array are meant to be exposed to the environment, a permutation network that puts the exposed terminals on a separate bus from the rest would be useful. To draw this distinction, we convert the above map relating terminal addresses with alphabet ordinals to a map

$$r = a^* \overset{\circ}{b}(\mu a^{-1})^* c \in (\mathbb{T} \cup \mathbb{V})^{|a|}$$

taking terminal addresses to transitions, and then divide r into two lists

$$\langle r \uparrow \mathcal{D}(A), r \uparrow \mathbb{T} - \mathcal{D}(A) \rangle \in (\mathbb{T} \cup \mathbb{V})^{*2}$$

where the first list $r \uparrow \mathcal{D}(A)$ contains transitions that are the origin of at least one member of the set A of arcs in the Petri net, and the second list $r \uparrow \mathbb{T} - \mathcal{D}(A)$ contains the rest. Terminals associated with transitions in the first list need to be connected to something else in the circuit, whereas the

rest correspond to exposed outputs. With the distinction thus drawn, we can convert each list of transitions back to a list of terminal addresses by writing

$$r^{-1**} \langle r \uparrow \mathcal{D}(A), r \uparrow \mathbb{T} - \mathcal{D}(A) \rangle \in \mathbb{N}^{*2}.$$

A permutation network based on the concatenation of these two lists certainly could route the exposed outputs to a separate bus from the rest, but the lines within each bus would still be out of order with respect to the alphabet. Whereas $r \uparrow \mathbb{T} - \mathcal{D}(A)$ lists the exposed output transitions in an order affected by their class membership, a list of these same transitions rearranged to

$$a \uparrow \mathcal{R}(r \uparrow \mathbb{T} - \mathcal{D}(A))$$

enumerates them in the order they appear in a . However, because r and a have the same ranges, this list might as well be expressed as

$$a \uparrow \mathbb{T} - \mathcal{D}(A)$$

and similarly for the other list, we have

$$a \uparrow \mathcal{D}(A)$$

suggesting a more useful choice of lists $l = (m_0 \ c) \ (a, A) \in \mathcal{P}(\mathbb{N})^{*2}$ for a permutation network by

$$m_0 = \lambda c. \lambda(a, A). (\lambda r. r^{-1**} \langle a \uparrow \mathcal{D}(A), a \uparrow \mathbb{T} - \mathcal{D}(A) \rangle) a^* \overset{\circ}{b} (\mu a^{-1})^* c \quad (16.18)$$

so that the first $|l_0|$ output terminals from the block $(\mathcal{F} \mathbf{R}) \ y \ \times \ b \ l$ are associated with the input or internal transitions in the Petri net, the remaining $|l_1|$ terminals are associated with the externally visible output transitions, and within each group they are ordered consistently with the alphabet ordering a .

Feedback FORK network

Getting from this point to the whole implementation of X is a matter of constructing some sort of a feedback path from the transitions implemented in $(\mathcal{F} \mathbf{R}) \ y \ \times \ b \ l$ to the inputs associated with the places. As well as a permutation network, a FORK network neglected until now is needed along the path. Any transition $t = a_k$ for any alphabet index $k \in \mathcal{D}(a)$ has a postset $q \ t \in \mathcal{P}(\mathbb{V})$ by

$$q = \Psi \Pi A \quad (16.19)$$

containing $|q \ a_k|$ places. To broadcast a signal to every place in its postset concurrently, the transition needs a back end block FORK $f \ k$ for $f = m_1(a, A) : \mathcal{D}(a) \rightarrow \mathbb{N}$ given by

$$m_1 = \lambda(a, A). (\lambda q. \lambda k. |q \ a_k|) \Psi \Pi A \quad (16.20)$$

with each of its $|q \ a_k|$ output terminals connected to an input terminal of one of the places in the postset. Incorporating the FORK networks leads to an adjustment along the lines of

$$(\mathcal{F} \mathbf{R}) \ y \xrightarrow{(b \ l)^{-1}} \mathbf{R}((\mathcal{F} \mathbf{R}) \ \text{FORK}^* \ f^* \ l_0 \ \times \ i, \mathbf{l}^{|l_1|})$$

subject to a permutation $i \in \mathbb{N}^*$ yet to be determined in preparation for a feedback bus $\mathbf{l}^{|i|}$ whereby we aim to write the whole solution as

$$\mathbf{Z}^{|i|} (\mathbf{Z}^{|i|} (\mathbf{R}((\mathcal{F} \mathbf{R}) \ y \xrightarrow{(b \ l)^{-1}} \mathbf{R}((\mathcal{F} \mathbf{R}) \ \text{FORK}^* \ f^* \ l_0 \ \times \ i, \mathbf{l}^{|l_1|}), \mathbf{l}^{|i|}) \downarrow |i|)).$$



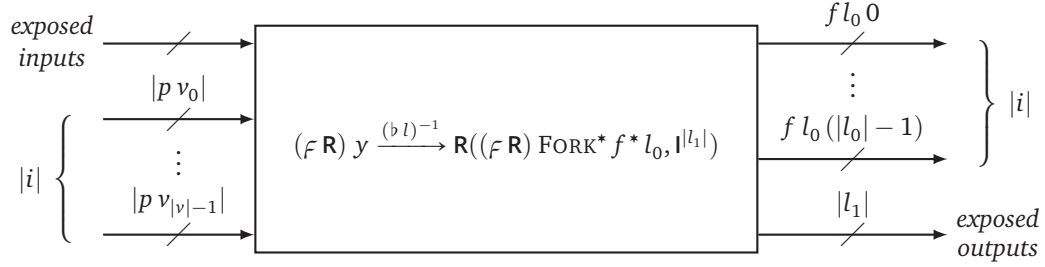


Figure 16.6: For the block to implement a Petri net with communities y , a permutation network with $|i|$ lines specified by a permutation i needs to interface the outputs from the transitions with the inputs to the places.

Feedback permutation network

To approach the permutation i , we envision the combined presets $p t$ by Equation 16.8 of all transitions $t \in u$ in each class $u \in \mathcal{R}(c)$ by Equation 16.17 listed in order of classes by

$$(\lambda u. \bigcup (\mu p) u)^* c \in \mathcal{P}(\mathbb{V})^*$$

giving rise to a flattened list of places

$$v = \overset{\circ}{b} (\lambda u. \bigcup (\mu p) u)^* c \in \mathbb{V}^*.$$

Each place $s \in \mathcal{R}(v)$ is implemented by a block internal to $(F R) y$ with one of $|p s|$ consecutive input terminals for each transition t in its preset $p s$. The places are grouped by classes and ordered lexicographically within each class by Equation 16.15. Their input terminals come last after the exposed input terminals as mandated by Equation 16.9, and the output terminals from the transitions as they appear prior to the incorporation of the feedback path are illustrated in Figure 16.6.

The output terminal connected to the input associated with a transition $t \in p s$ can be localized to the range connected to the output bus labeled $f l_0 a^{-1} t$ in Figure 16.6 because all lines emanating from the transition t are on this bus. Because there is one output bus for each transition, each bus's width is the cardinality of the corresponding transition's postset, and the bus's position relative to the other buses is the transition's position in a , the index of the terminal in question is at least the number

$$|\overset{\circ}{b} q^*(a \upharpoonright a^{-1} t)|$$

of bus lines due to all transitions preceding t in a by Equation 16.19. Furthermore, because the transition t may have other places than s in its postset, it reserves the $((q t)^\circ s)$ -th line on the bus for communication with s , implying the specific terminal index of

$$|\overset{\circ}{b} q^*(a \upharpoonright a^{-1} t)| + (q t)^\circ s.$$

The set of source terminal indices for all transitions $t \in p s$ is therefore the set

$$(\mu \lambda t. |\overset{\circ}{b} q^*(a \upharpoonright a^{-1} t)| + (q t)^\circ s) p s$$

leading to the list of these sets over all places $s \in \mathcal{R}(v)$ by

$$(\lambda s. (\mu \lambda t. |\overset{\circ}{b} q^*(a \mid a^{-1} t)| + (q t)^\circ s) p s)^* v \in \mathcal{P}(\mathbb{N})^*$$

and the desired permutation $i = m_2(c, p) (a, A) \in \mathbb{N}^*$ as given by

$$m_2 = \lambda(c, p). \lambda(a, A). (\lambda q. \overset{\circ}{b} (\lambda s. (\mu \lambda t. |\overset{\circ}{b} q^*(a \mid a^{-1} t)| + (q t)^\circ s) p s)^* (\lambda u. \bigcup (\mu p) u)^* c) \Psi \Pi A.$$

This result can be incorporated as planned into a result $m_3(y, \langle m_0 c, m_1, m_2(c, p) \rangle \triangle (a, A)^{\mathfrak{z}}) \in \mathbb{H}$ by

$$m_3 = \lambda(y, \langle l, f, i \rangle). \mathbf{Z}^{|i|} (\mathbf{Z}^{|i|} (\mathbf{R}((\mathcal{F} \mathbf{R}) y \xrightarrow{(\mathfrak{b} l)^{-1}} \mathbf{R}((\mathcal{F} \mathbf{R}) \text{FORK}^* f^* l_0 \times i, \mathbf{1}^{|l_1|}, \mathbf{1}^{|i|}) \downarrow |i|))$$

along with Equation 16.18 and Equation 16.20, or more explicitly into the combining form

$$\Omega_z^\alpha((T, A), y) = (\lambda \langle c, (a, p) \rangle. m_3(y, \langle m_0 c, m_1, m_2(c, p) \rangle \triangle (a, A)^{\mathfrak{z}})) (\langle \mathcal{U}_z^\alpha, \dot{\mathcal{U}}_z^\alpha \rangle \triangle (T, A)^{\mathfrak{z}}).$$

Mismatched alphabets

Before using the combining form Ω_z^α derived above to define the circuit $\text{DMS}_0(\alpha, X) \in \mathbb{H}$ in terms of a process $X = (I, O, N) \in \mathbb{D}$, we have to consider the issue of members of the alphabets I and O not appearing in the transitions T of the Petri net $N = (P, T, A, M, F)$. This issue is similar to one noted in Section 15.3.4, and much of the same discussion applies. The upshot is that a correct algorithm must add any input and output terminals to the circuit inferred from the Petri net as needed to match the nominal alphabets.

To distinguish between inputs $I \cap T$ present in T from inputs $I - T$ that are absent along with outputs $O \cap T$ and $O - T$, we can make a list

$$(\lambda s. \langle s \cap T, s - T \rangle)^* \langle I, O \rangle \in (\mathcal{P}(\mathbb{T})^2)^2$$

determining a record of their ordinals

$$(\mu (\alpha \uparrow (I \cup O))^{-1})^{**} (\lambda s. \langle s \cap T, s - T \rangle)^* \langle I, O \rangle \in (\mathcal{P}(\mathbb{N})^2)^2$$

sufficient for a function $m_4(\alpha, I, O) : \mathbb{H} \rightarrow \mathbb{H}$ to effect the required adjustment to the circuit by

$$m_4 = \lambda(\alpha, I, O). M_9 (\mu (\alpha \uparrow (I \cup O))^{-1})^{**} (\lambda s. \langle s \cap T, s - T \rangle)^* \langle I, O \rangle$$

for M_9 , given by Equation 15.25. With that, we have a specification for a function

$$\text{DMS}_0 : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$$

defining direct mapping synthesis by interacting state based synthetic communities as follows.

$$\text{DMS}_0(\alpha, X) = (\lambda(I, O, (P, T, A, M, F)). m_4(\alpha, I, O) \Omega_z^\alpha((T, A), \text{COM}_0(\alpha, A, M)^* \mathcal{U}_z^\alpha(T, A))) \mathbf{RP} X \quad (16.21)$$

16.6 Interacting direct mapped synthetic communities

When a process specification stubbornly resists decomposition into communities that can be implemented feasibly by state based synthesis, there is recourse to more drastic measures. Instead of

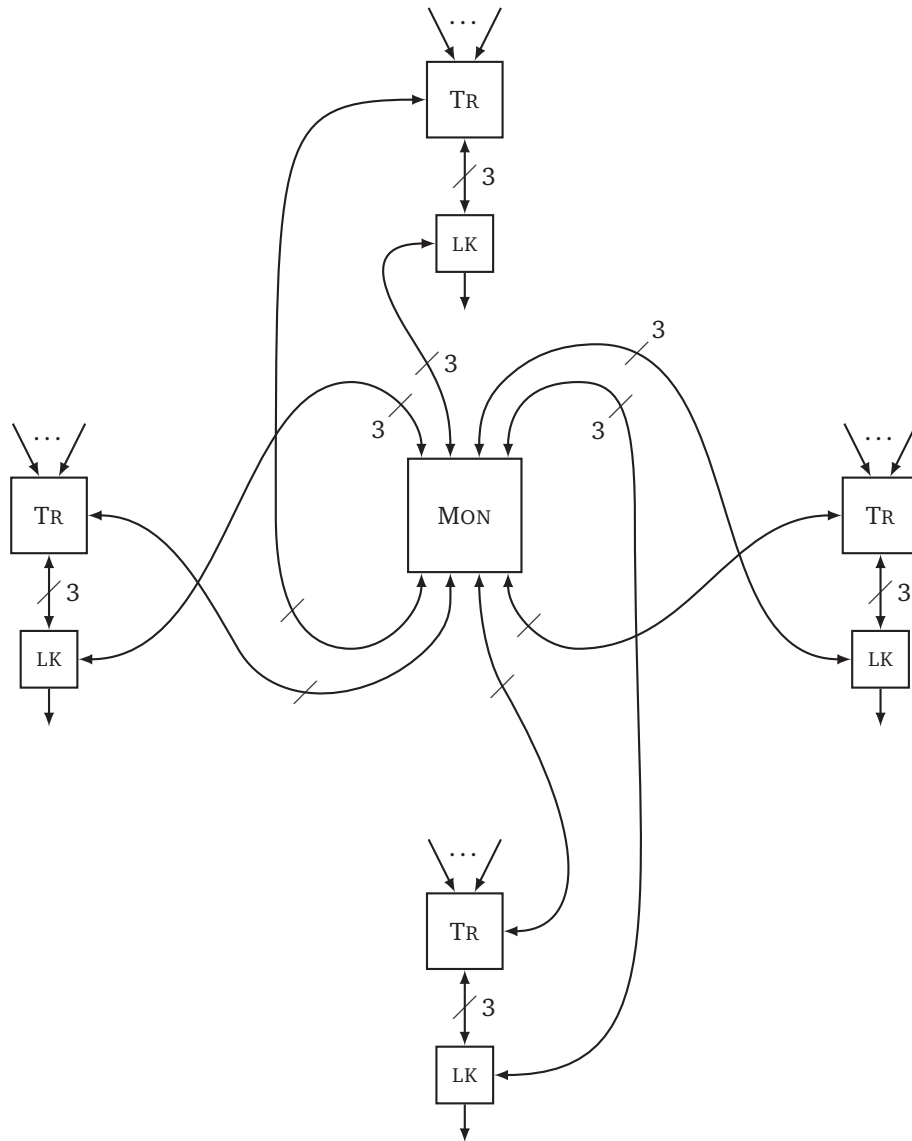


Figure 16.7: Transitions and a monitor in a direct mapped community interact via lock negotiation channels of three lines each and input revocation channels of various widths.

attempting to treat the whole community as a unit, we can treat each transition in it as a separate entity interacting independently with its environment. Synthesizing a circuit to behave as just one transition in a community is easier than synthesizing the whole community because the state space of an individual transition is smaller. Above a tipping point, it is easier to synthesize any number of transitions in a community separately than to synthesize them *en masse* because the work of synthesizing them separately increases only additively rather than multiplicatively with their number. Developing this idea informally at first and then making it more precise occupies the rest of this section.

16.6.1 Overview

Although each transition is to be implemented as a separate block according to this plan, some coordination among them is necessary to procure a faithful rendition of the process specification. If a transition receives a notification from every place in its preset indicating that the place holds a token, then that transition is ready to fire, but if any of those places belongs to the preset of another transition that is also ready to fire, then according to the specification at most one of the transitions is allowed to fire. The competition between the transitions has to be resolved somehow.

The monitor block shown at the center of [Figure 16.7](#) addresses this need. When a transition is ready to fire, it requests a lock from the monitor. If the monitor grants the lock, the transition fires, clears its record of available tokens, and releases the lock. The monitor keeps the system to its specification by never granting simultaneous locks to competing transitions.

Certain other aspects of the protocol between transitions and monitors necessarily ensue, which are worth sketching informally before getting down to business. If we envision the tokens being absorbed by the transition that fires, then the monitor must send a message to every other transition in receipt of the same notifications to inform them that the tokens are no longer available to be absorbed. For this discussion, these messages are called **revocation requests** because it is as if they revoke the notifications of token availability to the transitions.



- The monitor starts sending revocation requests the moment it decides to grant a lock but before granting it.
- Every revocation request from a monitor to a transition must be acknowledged by a **revocation acknowledgment** from the transition to the monitor.
- The monitor grants the lock only after receiving a revocation acknowledgment from every competing transition.

Due to unpredictable wire delays, a revocation request could reach a transition in advance of the notifications it revokes. The transition must be prepared for this possibility and block until the relevant notifications arrive if necessary.

What should happen if a transition sends a lock request to a monitor at the same time the monitor sends a revocation request to that transition? If the monitor is sending a revocation request, then some competing transition has already requested and been chosen to receive the lock, so the monitor certainly can not grant another lock. Nor can the monitor ignore the lock request because doing so allows the requesting transition to block, preventing it from acknowledging the pending revocation request and thereby deadlocking the system. A workable lock negotiation protocol therefore needs

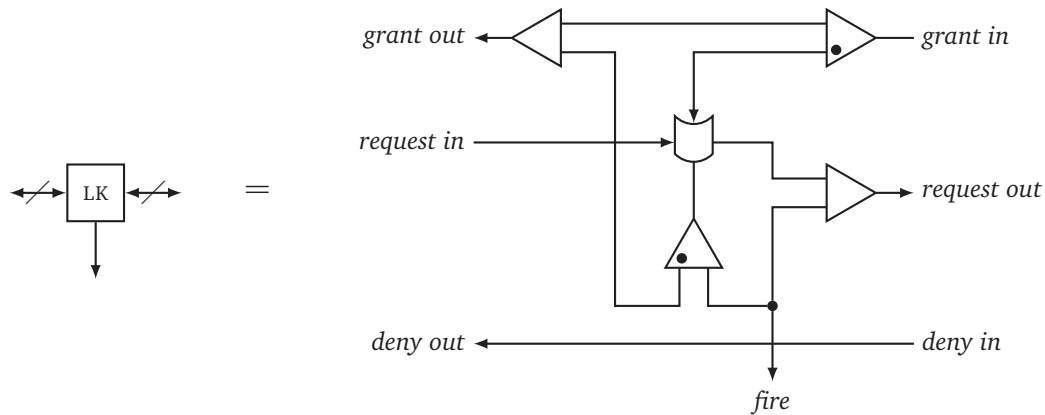


Figure 16.8: Locks are negotiated with a 2Φ request/deny handshake when requests are denied but with a 4Φ cycle for grants including an explicit release phase. The *fire* output is emitted concurrently with the release acknowledgment to signal that a transition fires when it releases a lock.

lock requests to be deniable explicitly, so that a transition having requested a lock blocks only until the request is either granted or denied.

- A monitor receiving a lock request from a transition to which it has sent a revocation request denies the lock request and continues waiting for the revocation acknowledgment.
- A transition whose lock request is denied can rely on a revocation request arriving shortly, so the transition waits for the next revocation request, acts on it accordingly, and does not request another lock until further notifications indicate a replenished supply of tokens.

In an obscure edge case, a speedy transition having been denied a lock waits for more notifications from its preset places, receives them, and requests another lock, all according to protocol and all before the competitor responsible for the denial has a chance to fire. Possibly the competitor has been selected to fire but not yet granted the lock owing to delayed revocation acknowledgments from other competitors. Granting the speedy transition's lock request would commit the monitor to initiating another round of revocation handshakes although the current round is still in progress. Denying it would condemn the speedy transition to wait forever for a revocation request that never comes. In this case the monitor takes the easy way out by postponing any grant or denial decision until the pending cycle concludes with a lock release. The speedy transition is temporarily blocked, but there is no danger of deadlock. The monitor can always distinguish between deniable and blockable lock requests because the latter occur only during the interval between a revocation acknowledgment from the requesting transition and a lock release from its competitor. Notably this solution depends on locks being released explicitly, mandating a 4Φ handshake for successful lock negotiation as implied above, even if a 2Φ handshake suffices for denials.

16.6.2 Transitions

Designing transition and monitor blocks down to the level of individual primitive components that execute this protocol may seem like a daunting prospect, but will yield in due course to an approach

following the example of a certain mad scientist mentioned in [Section 16.2](#). In this section we focus on the transition block, which is the easier of the two, and make it slightly easier by delegating a small part of its functionality to the manually designed block shown in [Figure 16.8](#).

Bus interfaces

The transition block is best viewed as interfacing with its environment by way of three buses operating concurrently. One of the buses carries token availability notifications from the preset places, and has one input for each member of the transition's preset. Another bus carries the lock negotiation channel over three lines. One line of the lock negotiation channel is an output from the transition to request or release a lock, and the other two are inputs whereby the transition is either granted or denied a lock. The remaining bus communicates revocation requests and acknowledgments. The same output from the transition is used to acknowledge any revocation request, but there are multiple inputs, each corresponding to a specific set of token notifications to be revoked. There is no dedicated output to signal that the transition fires. Instead we envision the LK block shown in [Figure 16.8](#) wire tapping the lock negotiation channel to emit a firing signal whenever the transition releases a lock.

A more precise account of this interface requires further details about the revocation channel, specifically as to the number of inputs and the set of notifications associated with each. A transition $t \in u$ in a class $u \in \mathcal{U}_z^\alpha(T, A)$ of the partition $\mathcal{U}_z^\alpha(T, A)$ derived from some process whose Petri net model has transitions T and arcs A has a preset $\bullet t = p t$ by [Equation 16.8](#), so it has $|\mathcal{P}(p t)|$ subsets of notification inputs, but fortunately never needs more than the greater of $|u| - 1$ or $2^{|p t|} - 1$ revocation inputs, and often needs fewer. The exact number $|j|$ is determined by the set j of non-empty intersections between $p t$ and $p w$ over all other transitions $w \in u - \{t\}$.

$$j = \bigcup_{w \in u - \{t\}} \{(p t) \cap p w\} - \{\emptyset\} \in \mathcal{P}(\mathcal{P}(\mathbb{V})) \quad (16.22)$$

When a transition w other than t is granted a lock, the monitor calls for t to revoke exactly the notifications from places in $(p t) \cap p w$, and these sets of notifications are the only possibilities. As for the correspondence between revocation bus inputs and sets of preset places, we can identify the n -th bus line with the lexicographically n -th term $(j^{\circ-1})_n$ for $0 \leq n < |j|$ based on the implicit total ordering of \mathbb{V} .

Parameterized processes

This description of the interface along with that of the protocol in [Section 16.6.1](#) is enough to get started with constructing the circuit to implement a transition by state based synthesis as planned. Although a manual design for an individual transition might be contemplated, there is no one size that fits all combinations of presets and competitors, leaving little alternative but to derive a circuit algorithmically from a process parameterized by them. Assuming an input alphabet using the first $|j|$ generic terminal symbols by [Equation 16.22](#) to represent the revocation inputs and the next $|p t|$ generic terminal symbols to represent the token notification inputs, we can write

$$i = f_2(p, u, t) \in \mathcal{P}(\mathcal{P}(\mathbb{G})) \quad (16.23)$$

temporarily for a set of sets of token notification symbols wherein each set represents the inputs revoked by some particular revocation symbol in terms of the preset function p , the class of transitions

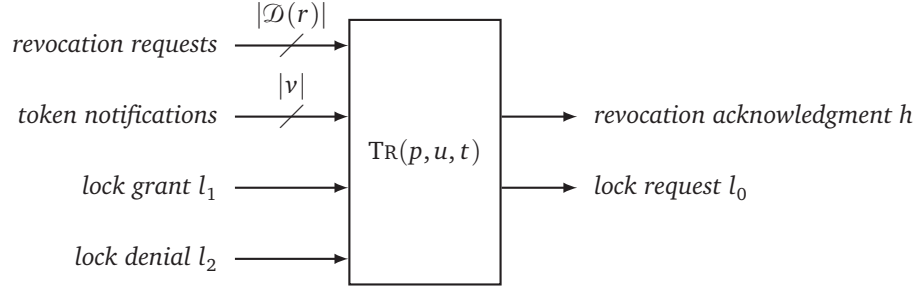


Figure 16.9: transition process interface for parameters $(v, h, l, r) = f_3(p, u, t) f_2(p, u, t)$ with terminals shown in lexicographic order of their associated alphabet symbols

u , the transition $t \in u$ whose implementation is sought, and the function

$$f_2 = \lambda(p, u, t). (\lambda j. (\mu^2 (\lambda k. \mathbb{G}^{\circ-1} |j| + (p t)^\circ k)) j) \bigcup_{w \in u - \{t\}} \{(p t) \cap p w\} - \{\emptyset\}. \quad (16.24)$$

Then we can write

$$v = \mathcal{R}(\mathbb{G}^{\circ-1*} \iota_{|d|}^{|i|}) \in \mathcal{P}(\mathbb{G})$$

explicitly for the subset of input alphabet symbols corresponding to token notifications in terms of the preset $d = p t$, take the next unused generic symbol

$$h = \mathbb{G}^{\circ-1} |i| + |d|$$

for the revocation acknowledgment output, and reserve a list of three more distinct alphabet symbols

$$l = \mathbb{G}^{\circ-1*} \iota_3^{|i|+|d|+1} \in \mathbb{G}^3$$

for the lock channel. To associate a set $s \in \mathcal{P}(v)$ of token notification input symbols with its revocation input, a function

$$r = \mathbb{G}^{\circ-1} \circ i^\circ : \mathcal{P}(v) \rightarrow \mathbb{G}$$

makes it easy to write $r(s) \in \mathbb{G}$ provided s is a member of i by Equation 16.23, meaning the inputs in s are revocable as a unit. Henceforth we identify the domain $\mathcal{D}(r)$ with the set i rather than referring to i explicitly. A tuple

$$(v, h, l, r) = f_3(p, u, t) f_2(p, u, t) \quad (16.25)$$

summarizes this information for any transition t with p and u as above in terms of a function

$$f_3 = \lambda(p, u, t). \lambda i. (\lambda d. (\mathcal{R}(\mathbb{G}^{\circ-1*} \iota_{|d|}^{|i|}), \mathbb{G}^{\circ-1} |i| + |d|, \mathbb{G}^{\circ-1*} \iota_3^{|i|+|d|+1}, \mathbb{G}^{\circ-1} \circ i^\circ)) p t. \quad (16.26)$$

This summary of the transition process alphabets facilitates an expression of the process by a dependence graph representing a system of recurrences as in Figure 16.3, which is then solvable for a single process by the generalized fixed point combinator Υ defined in Equation 16.5. This process is then transformable by state based synthesis to the block shown in Figure 16.9 with the lexicographic alphabet ordering $f_4(p, u, t) f_2(p, u, t) \in \mathbb{G}^*$ by

$$f_4 = \lambda(p, u, t). \lambda i. (\lambda k. \mathbb{G}^{\circ-1*} \iota_k) |i| + |p t| + 4. \quad (16.27)$$

Dependence graphs

While the generalized fixed point combinator requires the edges in the dependence graph to be labeled by lists of processes, the vertices as noted previously can be drawn from any totally ordered set, so we are free to choose the vertices in a way that makes the adjacency relation obvious. A good choice for the current setting would be a graph of sets of token notification input symbols, with each vertex in the graph corresponding to one of an ensemble of processes such that a vertex s corresponds to the process that has already received all token notifications in s . Then for example we would know that any vertex $s \subset v$ must have an outgoing edge labeled $\langle \text{get } i \rangle$ for any token notification input symbol $i \in v - s$, and the edge leads obviously to the vertex $s \cup \{i\}$. The intuition is that the process corresponding to the vertex $s \cup \{i\}$ behaves slightly differently from the one corresponding to s in a way that reflects having received the input i .

It is tempting to continue immediately in this vein to build the whole dependence graph, which involves only three other kinds of edges than the one in the example above, but there is a small issue. The solution obtained from the dependence graph by the generalized fixed point combinator is invariably an open Petri net modeled process. Building the reachability graph of an open Petri net in the course of state based synthesis can be considerably more costly than building one for the equivalent closed Petri net, and even prohibitive. Typically this cost is mitigated by specifying an environment in combination with the process to obtain a closed Petri net. Whereas the environment is usually simple enough to be an afterthought, specifying a suitable environment to interact with the transition process is comparable to specifying the transition process itself. In anticipation of this issue, we can construct an environment of the same form as the process by reflecting the **get** operations in each of them as **put** operations in the other in terms of a function

$$\tilde{\tau}: (\mathcal{P}(\mathbb{G}) \times \mathbb{G} \times \mathbb{G}^3 \times (\mathcal{P}(\mathbb{G}) \rightarrow \mathbb{G})) \times (\mathbb{T} \rightarrow \mathbb{D})^2 \rightarrow (\mathcal{P}(\mathbb{G}) \rightarrow \mathcal{P}(\mathbb{D}^* \times \mathcal{P}(\mathbb{G})))$$

parameterized by the tuple

$$(v, h, l, r) \in \mathcal{P}(\mathbb{G}) \times \mathbb{G} \times \mathbb{G}^3 \times (\mathcal{P}(\mathbb{G}) \rightarrow \mathbb{G})$$

describing the alphabets by [Equation 16.25](#), and by either of the lists

$$\langle \text{get}, \text{put} \rangle, \langle \text{put}, \text{get} \rangle \in (\mathbb{T} \rightarrow \mathbb{D})^2$$

so that $\tilde{\tau}((v, h, l, r), \langle \text{get}, \text{put} \rangle)$ s defines an adjacency set in $\mathcal{P}(\mathbb{D}^* \times \mathcal{P}(v))$ of a vertex $s \in \mathcal{P}(v)$ in the dependence graph of the transition process, but $\tilde{\tau}((v, h, l, r), \langle \text{put}, \text{get} \rangle)$ s' uses the same function $\tilde{\tau}$ to define the adjacency set of a vertex s' in the dependence graph of the environment of the transition. We can interpret s' as the vertex corresponding to the one of an ensemble of mutually dependent processes making up the environment that has already sent all token notifications in s' to the transition.

An adjacency set $(\tilde{\tau} z)$ s for $z = ((v, h, l, r), \langle g, p \rangle)$ associated with a vertex s in a dependence graph is convenient to express as the union of four sets $(\tilde{\tau}_k z)$ s of edges in terms of functions $\tilde{\tau}_0$ through $\tilde{\tau}_3$ according to these definitions.

- Any vertex $s \subset v$ that does not contain the complete set v of token notifications has an adjacency set containing at least the members of $(\tilde{\tau}_0 z)$ s as noted previously for $\tilde{\tau}_0$ defined as follows.

$$\tilde{\tau}_0 = \lambda((v, h, l, r), \langle g, p \rangle). \lambda s. \bigcup_{i \in v-s} \{(\langle g \ i \rangle, s \cup \{i\})\}$$

- Any vertex $s \subset v$ corresponding to a process having received (or an environment having sent) some revocable set $b \in \mathcal{D}(r)$ of token notifications, but not the complete set v , can participate in a revocation handshake leading to a vertex $s - b$. This handshake is initiated by the revocation request $r b \in \mathbb{G}$ and completed by the revocation acknowledgment $h \in \mathbb{G}$ according to \bar{t}_1 defined as follows.

$$\bar{t}_1 = \lambda((v, h, l, r), \langle g, p \rangle). \lambda s. \bigcup_{b \in (\mu r)(\mathcal{D}(r) \cap \mathcal{P}(s))} \langle \langle g r b, p h \rangle, s - b \rangle, \emptyset \rangle_{\delta_s^v}$$

- A vertex $s = v$ corresponding to the process having received (or sent) every token notification in v can participate in a successful 4Φ lock negotiation and clear its record of notifications using l_0 as the lock request signal and l_1 as the grant by this definition of \bar{t}_2 .

$$\bar{t}_2 = \lambda((v, h, l, r), \langle g, p \rangle). \lambda s. \langle \emptyset, \{ \langle p l_0, g l_1, p l_0, g l_1 \rangle, \emptyset \} \rangle_{\delta_s^v}$$

- A vertex $s = v$ also has outgoing edges in $(\bar{t}_3 z) s$ representing participation in an unsuccessful 2Φ lock negotiation followed by a revocation handshake for any set of notifications $b \in \mathcal{D}(r)$, leading to a vertex $s - b$ by this definition of \bar{t}_3 .

$$\bar{t}_3 = \lambda((v, h, l, r), \langle g, p \rangle). \lambda s. \bigcup_{b \in \mathcal{D}(r)} \langle \emptyset, \{ \langle p l_0, g l_2, g r b, p h \rangle, s - b \} \rangle_{\delta_s^v}$$

Transition block synthesis

These definitions for \bar{t}_0 through \bar{t}_3 enable an expression $q(s)$ for the pair of a vertex s and its associated adjacency set in the transition process dependence graph by

$$q(s) = \left\{ \left(s, \bigcup_{k=0}^3 \bar{t}_k(w, \langle \mathbf{get}, \mathbf{put} \rangle) s \right) \right\}$$

and the whole process by the generalized fixed point combinator and a percolation of q from the empty vertex.

$$\Upsilon (\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} q n) q \emptyset$$

However, to express either the process or its reflecting environment with equal ease, we can have a direction $d \in \{0, 1\}$ induce either the process by $d = 0$ or the environment by $d = 1$ in an expression

$$\Upsilon (\lambda q. (\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} q n) q \emptyset) \lambda s. \left\{ \left(s, \bigcup_{k=0}^3 \bar{t}_k(w, \langle \mathbf{get}, \mathbf{put} \rangle \circ \langle \delta_1^d, \delta_0^d \rangle) s \right) \right\}$$

or express both at once as $f_5 w \in \mathbb{D}^2$ by

$$f_5 = \lambda w. (\lambda d. \Upsilon (\lambda q. (\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} q n) q \emptyset) \lambda s. \left\{ \left(s, \bigcup_{k=0}^3 \bar{t}_k(w, \langle \mathbf{get}, \mathbf{put} \rangle \circ \langle \delta_1^d, \delta_0^d \rangle) s \right) \right\}^* \iota_2$$

which along with Equation 16.24, Equation 16.26, and Equation 16.27 leads to a complete description of the block shown in Figure 16.9 by state based synthesis.

$$\text{TR}(p, u, t) = (\lambda \langle i, g, f \rangle. \text{SBS}(g i, (\mathcal{F} \text{env}) f i)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle (p, u, t)^{\exists}) \quad (16.28)$$

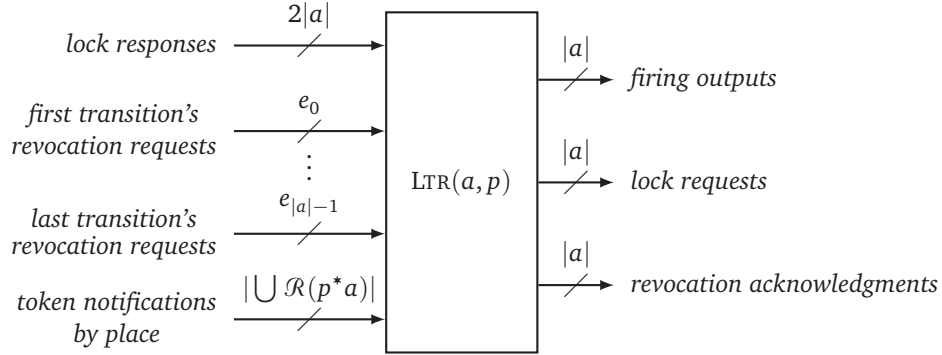


Figure 16.10: A lockable transition array parameterized by $(a, p) = \dot{\mathcal{U}}_z^\alpha(u, A)$ for a set $u \in \mathcal{U}_z^\alpha(T, A)$ of transitions determined by a Petri net with transitions T and arcs A combines $|a| = |u|$ transition blocks, where e_n is the number of revocation requests for the n -th transition in the array.

16.6.3 Lockable transitions

Combining the transition blocks defined in Section 16.6.2 with a monitor block foreshadowed in Section 16.6.1 and soon to be defined in Section 16.6.4 benefits from an intermediate step in this section to construct the block shown in Figure 16.10, mnemonic for “lockable transition array”. Parameterized by a preset function and alphabet ordering $(a, p) = \dot{\mathcal{U}}_z^\alpha(u, A)$ by Equation 16.10, this block hides an array of transition blocks behind a FORK network having a single input for each place in their collective presets $\bigcup \mathcal{R}(p^*a)$. The FORK network carries a token notification signal from each place to all of the transitions in the place’s postset. Each transition is also connected to an instance of the block LK shown in Figure 16.8. Finally, permutation networks on the front and back organize the rest of the signals into separate buses for simpler coordination.

Lock blocks

To work from the inside out, we can start with a block combinator expression

$$\text{LK} = \mathbf{R}(\mathbf{Z}^3(\langle \mathbf{Z}^2 \mathbf{R}(\mathbf{F}_2 \langle \text{TOGGLE}^2 \uparrow 1, \text{MERGE}^2 \rangle, \mathbf{R}(\text{SHUNT}, \text{FORK}) \downarrow 1) \rangle \downarrow 1 \uparrow 2), 1)$$

for LK derived as shown in Figure 16.11. Consequently, the first input is a lock request from the transition, the next input is a grant from the monitor, and the next input is a denial from the monitor. The output side starts with a firing output, followed by a lock grant output destined for the transition, then a lock request output destined for the monitor, and then a lock denial output for the transition. Based on this ordering and the one shown in Figure 16.9, the three connections between the LK and a block derived from a transition $t \in u$ are effected by

$$\mathbf{Z}(\mathbf{L}_2 \langle \text{LK} \uparrow_2^1, \text{TR}(p, \mathcal{R}(a), t) \rangle \uparrow 1)$$

with the range $\mathcal{R}(a) = u$ being the set of transitions in the community. The combined block has two lock acknowledgment inputs followed by a revocation request bus followed by a token notification bus as inputs, with a firing output, lock request, and revocation acknowledgment as outputs.

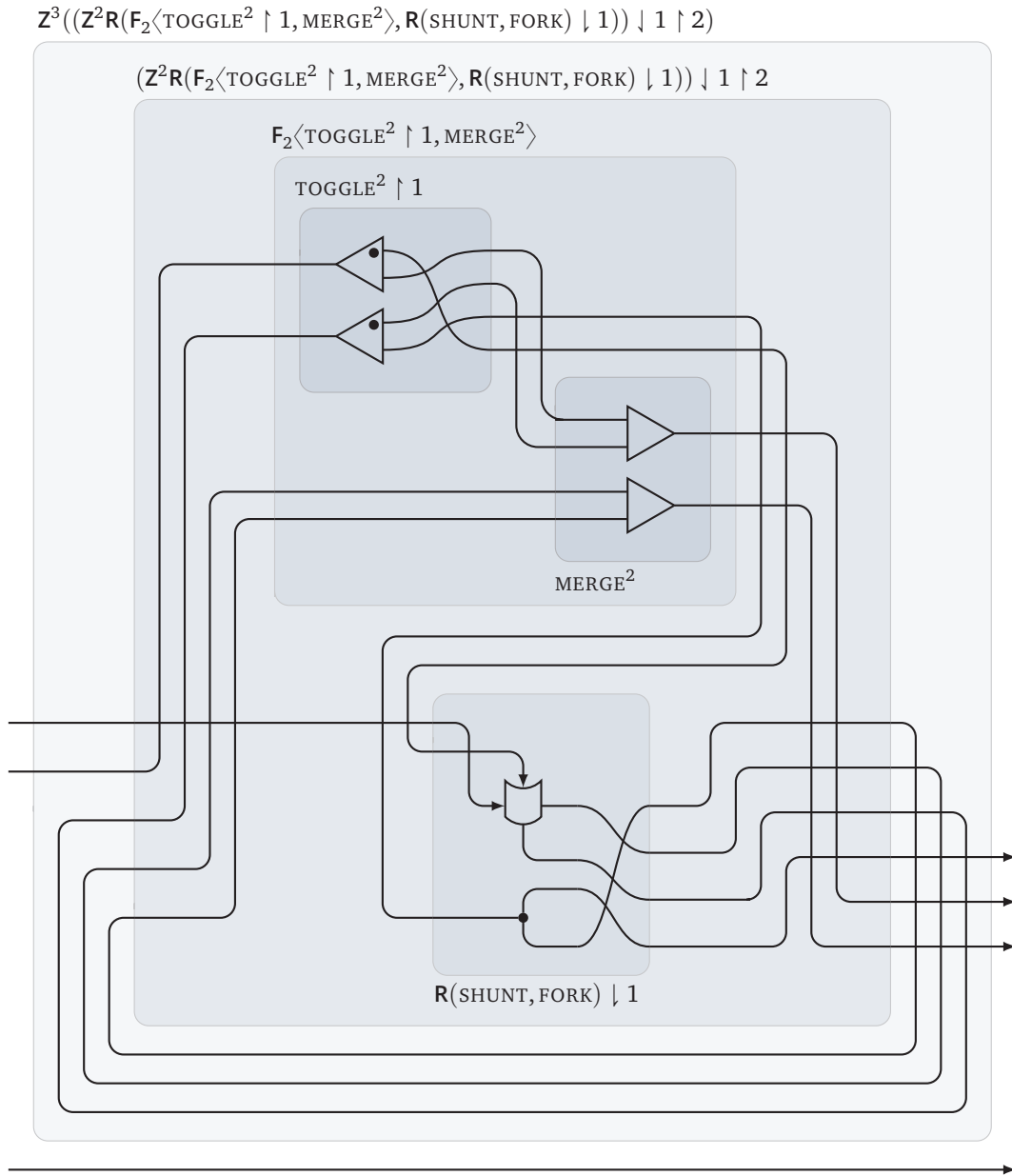


Figure 16.11: A block $R(Z^3((Z^2R(F_2\langle TOGGLE^2 \uparrow 1, MERGE^2 \rangle), R(SHUNT, FORK) \downarrow 1)) \downarrow 1 \uparrow 2), 1)$ to express the circuit in Figure 16.8 has inputs in order of request, grant and deny signals, and has outputs ordered as fire, grant, request, and deny.

Parallel transitions

The input and output ordering on a single transition block combined with LK would be consistent with Figure 16.10 if they were the only ones, but further effort would be needed if there were blocks of this form for multiple transitions in parallel. A partial solution is to combine the blocks in a fold

$$(\mathcal{F} \lambda(t, z). \mathbf{R}(\mathbf{Z}(\mathbf{L}_2 \langle \mathbf{LK} \Gamma_2^1, \text{TR}(p, \mathcal{R}(a), t) \rangle \uparrow \mathbf{1}) \downarrow |p t|, z) \uparrow |p t|) a$$

so that the token notification bus inputs come last in reverse order by transitions but still in the original order of preset places on the bus for each transition. We can at least also finish with the output buses by writing

$$((\mathcal{F} \lambda(t, z). \mathbf{R}(\mathbf{Z}(\mathbf{L}_2 \langle \mathbf{LK} \Gamma_2^1, \text{TR}(p, \mathcal{R}(a), t) \rangle \uparrow \mathbf{1}) \downarrow |p t|, z) \uparrow |p t|) a) \times \iota_{3|a|} \times |a|$$

to route the firing outputs, lock requests, and revocation acknowledgments onto three separate buses as shown in Figure 16.10 (by Equation 8.12), and abbreviate this expression as $f_6(a, p)$ (TR, LK) for

$$f_6 = \lambda(a, p). (\lambda(i, l). (\mathcal{F} \lambda(t, z). \mathbf{R}(\mathbf{Z}(\mathbf{L}_2 \langle \mathbf{L} \Gamma_2^1, i(p, \mathcal{R}(a), t) \rangle \uparrow \mathbf{1}) \downarrow |p t|, z) \uparrow |p t|) a) \times \iota_{3|a|} \times |a|. \quad (16.29)$$

Input permutation network

This array leaves the lock response inputs interspersed with the revocation request inputs unlike Figure 16.10 unless a permutation network untangles them, but a permutation given similarly to the output permutation by Equation 8.12 is inadequate because the revocation request bus widths vary with the transition. In particular, a block derived from a transition $t \in \mathcal{R}(a)$ has

$$\left| \bigcup_{v \in (\mu p)(\mathcal{R}(a) - \{t\})} \{v \cap p t\} - \{\emptyset\} \right|$$

revocation inputs, one for each non-empty intersection $v \cap p t$ of the preset $p t$ with the preset v of some other transition in $\mathcal{R}(a)$, and generally the n -th transition block in the array has e_n revocation request inputs for $0 \leq n < |a|$ and

$$e = (\lambda t. \left| \bigcup_{v \in (\mu p)(\mathcal{R}(a) - \{t\})} \{v \cap p t\} - \{\emptyset\} \right|)^* a$$

(cf. Equation 16.22). Because the n -th transition block accounts for two lock response inputs followed by e_n revocation request inputs, the two lock response inputs for the n -th transition block appear at positions

$$\iota_2^{2n + \sum(e_i n)}$$

on the array, and the e_n revocation request inputs at positions

$$\iota_{e_n}^{2 + 2n + \sum(e_i n)}.$$

A list of the lock response input terminal positions followed by a list of the revocation request input terminal positions therefore could be expressed

$$d = \flat \flat^* ((\lambda n. \langle \iota_2^{2n + \sum(e_i n)}, \iota_{e_n}^{2 + 2n + \sum(e_i n)} \rangle)^* \iota_{|e|})^\top \quad (16.30)$$

using the transpose notation defined in [Section 11.1.2](#), and is useful enough for specifying a front end permutation network to be worth abbreviating as $d = f_7(a, p)$ by

$$f_7 = \lambda(a, p). (\lambda e. \mathring{b} \mathring{b}^* ((\lambda n. \langle t_2^{2n+\Sigma(e_1 n)}, t_{e_n}^{2+2n+\Sigma(e_1 n)} \rangle)^* \iota_{|e|})^\top) (\lambda t. |\bigcup_{v \in (\mu p) (\mathcal{R}(a) - \{t\})} \{v \cap p t\} - \{\emptyset\}|)^* a.$$

FORK network

For the rest of the inputs to the array, which correspond to the token notification inputs on the transition blocks, we have to ensure the notification from each place $v \in \bigcup \mathcal{R}(p^* a)$ in the collective preset reaches every transition in the postset $q = \{i \in \mathcal{R}(a) \mid v \in p i\}$ of v through a FORK with $|q|$ outputs. Each output from the FORK associated with the place v is destined for a token notification input on a transition block corresponding to a transition $t \in q$, specifically the token notification input in the position

$$(p t)^\circ v$$

relative to others connected to the same transition block, because locally they are lexicographically ordered by the places sending them. The position relative to token notification inputs on all blocks would be offset to

$$|\mathring{b} p^* (a \ll (a^{-1} t) + 1)| + (p t)^\circ v$$

because the blocks following that of t in the array have their notification inputs preceding it for reasons noted in connection with [Equation 16.29](#). Furthermore, an offset of $|d|$ by [Equation 16.30](#) applies to all of them because they come after the lock response and input revocation buses as shown in [Figure 16.10](#). Hence for a given preset place v , the set of input terminal positions connected to its FORK outputs is

$$\bigcup_{t \in \{i \in \mathcal{R}(a) \mid v \in p i\}} \{|d| + |\mathring{b} p^* (a \ll (a^{-1} t) + 1)| + (p t)^\circ v\}$$

and a list $f_8(a, p) d \in \mathcal{P}(\mathbb{N})^*$ of these sets of input terminal positions in order of their associated places is given by

$$f_8 = \lambda(a, p). \lambda d. (\lambda v. \bigcup_{t \in \{i \in \mathcal{R}(a) \mid v \in p i\}} \{|d| + |\mathring{b} p^* (a \ll (a^{-1} t) + 1)| + (p t)^\circ v\})^* (\bigcup \mathcal{R}(p^* a))^{-1}.$$

Interface

This list determines a permutation $d \mathring{b} f_8(a, p) d$ and a network of blocks FORK $|(f_8(a, p) d)_i|$ for each preset place numbered $0 \leq i < |f_8(a, p) d|$ to interface with the array of transition blocks in a definition of the lockable transition array given by

$$\text{LTR}(l) = (\lambda \langle h, d, f \rangle. (\mathcal{F} \mathbf{R}) (|\mathring{d}| : (\lambda i. \text{FORK } |(f d)_i|)^* \iota_{|f d|}) \xrightarrow{d \mathring{b} f d} h(\text{TR}, \text{LK})) (\langle f_6, f_7, f_8 \rangle \triangle l^{\mathring{3}}). \quad (16.31)$$

16.6.4 Monitors

The purpose of the monitor block shown in [Figure 16.12](#) is to arbitrate among the transitions in the lockable transition array shown in [Figure 16.10](#) to ensure that it faithfully simulates the flow of tokens through the Petri net model of a process. The inputs and outputs on these blocks complement

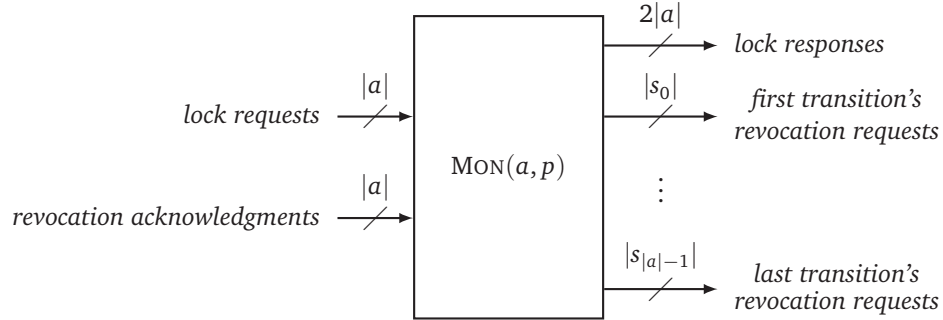


Figure 16.12: A monitor parameterized by (a, p) as in Figure 16.10 is designed to interface with a lockable transition array partly by $|a|$ buses of widths $|s_n| = e_n$, where $s_n \in \mathcal{P}(\mathcal{P}(\mathbb{V}))$ specifies a set of sets of places the preset of the n -th transition in the array has in common with those of others.

one another to the extent that the monitor becomes hidden from the environment with only the token notification inputs and firing outputs left exposed on the lockable transition array when they are combined as in Section 16.6.5.

Whereas a community $u \in \mathcal{U}_z^\alpha(T, A)$ in a decomposition $\mathcal{U}_z^\alpha(T, A)$ derived from a process specification contains $|u|$ transitions, there is at most one monitor for the whole community. Nevertheless, monitor blocks can vary depending on the size and organization of the community in a way that compels an algorithmic approach to their synthesis from adjustable parameters, similarly to the synthesis of transition blocks in Section 16.6.2.

Competitors

Starting from a pair of parameters $(a, p) = \dot{\mathcal{U}}_z^\alpha(u, A)$ with a list a of $|u|$ transitions ordered by α and a function p mapping each Petri net vertex to its preset by Equation 16.10, we seek a description of a monitor block parameterized in part by a list $c \in \mathcal{P}(\mathbb{N})^{|a|}$ with

$$c = (\lambda t. (\mu a^{-1}) \bigcup_{v \in p t} \{l \in \mathcal{R}(a) - \{t\} \mid v \in p l\})^* a \quad (16.32)$$

denoting a list of sets of transition indices relative to a such that c_n is the set of indices of transitions that compete with the n -th in the sense that their presets intersect. If the monitor is to grant a lock to the n -th transition, then it must deny lock requests from transitions numbered among c_n .

Revocation channels

The description of the monitor also depends more visibly on a list $s \in \mathcal{P}(\mathcal{P}(\mathbb{V}))^{|a|}$ associating a transition $k = a_n$ with the set $s_n \in \mathcal{P}(\mathcal{P}(\mathbb{V}))$ of subsets of its preset $p k \in \mathcal{P}(\mathbb{V})$ that it has in common with other transitions by

$$s = (\lambda k. \bigcup_{j \in \mathcal{R}(a) - \{k\}} \{(p k) \cap p j\})^* a.$$

The utility of this list is in identifying which of the $|s_n|$ revocation inputs to the n -th transition should be signaled when a lock is to be granted to the t -th transition. If the intersection $s_n \cap s_t$ contains

only one member $x \in s_n \cap s_t$, then the $((s_n)^\circ x)$ -th revocation input is the only choice, but more generally, the presets of both the n -th and the t -th transition could have the same intersection y with that of a third transition, and therefore have both of $x, y \in s_n \cap s_t$ in common with each other. However, if x and y are not equal, then one must be a proper subset of the other and only the larger of the two can contain all places common to the presets of both the n -th and the t -th transition. Because the n -th transition must revoke all notifications from places common to its preset and that of the t -th, the appropriate revocation signal is given by

$$(s_n)^\circ (\max(\mu \lambda x. \langle |x|, x \rangle) (s_n \cap s_t))_1$$

which is the one corresponding to the intersection of maximum cardinality. To summarize up to this point, we have a list $\langle c, s \rangle = g_0(a, p)$ of two lists by

$$g_0 = \lambda(a, p). \langle (\lambda t. (\mu a^{-1}) \bigcup_{v \in p t} \{l \in \mathcal{R}(a) - \{t\} \mid v \in p l\})^*, (\lambda k. \bigcup_{j \in \mathcal{R}(a) - \{k\}} \{(pk) \cap p j\})^* \rangle \triangle a^2 \quad (16.33)$$

with the list $c \in \mathcal{P}(\mathbb{N})^{|a|}$ of competitor sets to be of further interest shortly, and the list s whereby we can identify precisely the revocation request required of the n -th transition indexed relative to a when a lock is awarded to any other.

Alphabet assignments

As a step further toward describing the monitor block, the revocation request buses regarded collectively as a single bus ordered as shown in [Figure 16.12](#) imply a revocation output position

$$|\overset{\circ}{b}(s \mid n)| + (s_n)^\circ (\max(\mu \lambda x. \langle |x|, x \rangle) (s_n \cap s_t))_1$$

relative to the collective bus, which incorporates the offset $|\overset{\circ}{b}(s \mid n)|$ due to buses prior to the n -th, with s, n , and t interpreted as above. Moreover, if the monitor block is obtained by state based synthesis from the alphabet ordering $g_1 g_0(a, p) \in \mathbb{G}^*$ by

$$g_1 = \lambda \langle c, s \rangle. (\lambda i. \mathbb{G}^{\circ^{-1} * } \iota_i) 4|c| + |\overset{\circ}{b}s| \quad (16.34)$$

containing the first $4|c| + |\overset{\circ}{b}s|$ generic symbols as needed to accommodate the buses in [Figure 16.12](#), and the symbols are assigned to the buses consistently with the order illustrated, then a list

$$r = g_2 s \in \mathcal{P}(\mathbb{G})^{|a|} \quad (16.35)$$

can be made to enumerate the set $r_t \in \mathcal{P}(\mathbb{G})$ of revocation request output alphabet symbols to be emitted when the transition a_t is granted a lock by

$$g_2 = \lambda s. (\lambda t. \bigcup_{n \in \{i \in \mathcal{D}(s) - \{t\} \mid s_i \cap s_t \neq \emptyset\}} \{\mathbb{G}^{\circ^{-1} } 3|s| + |\overset{\circ}{b}(s \mid n)| + (s_n)^\circ (\max(\mu \lambda x. \langle |x|, x \rangle) (s_n \cap s_t))_1\})^* \iota_{|s|}. \quad (16.36)$$

The offset $3|s| = 3|a|$ to each symbol ordinal in r is to avoid clashing with generic symbols

$$l = \mathbb{G}^{\circ^{-1} ** } (\lambda i. \iota_3^{3i})^* \iota_{|r|}$$

representing the lock channels at the beginning of the range as a list of $|r| = |s| = |a|$ lists of three symbols each, which leaves only the list

$$h = \mathbb{G}^{\circ-1*} \iota_{|r|}^{3|r|+|b|r|}$$

of alphabet symbols at the end of the range for the revocation acknowledgments. Together with the list c from Equation 16.32, the list of parameters

$$\langle c, r, l, h \rangle = c : g_3 r \quad (16.37)$$

by a definition of

$$g_3 = \lambda r. r : \langle \mathbb{G}^{\circ-1**} (\lambda i. \iota_3^{3i})^* \iota_{|r|}, \mathbb{G}^{\circ-1*} \iota_{|r|}^{3|r|+|b|r|} \rangle \quad (16.38)$$

enables a simpler description of the monitor as a process than one expressed directly in terms of the original parameters (a, p) .

Dependence graphs

Similarly to the transition process specification in Section 16.6.2, a monitor process fit for state based synthesis is manageable as a dependence graph with vertices chosen to elucidate the adjacency relation. As in previous examples, the dependence graph induces a system of recurrences solvable for the desired process by the generalized fixed point combinator Υ defined in Equation 16.5. Whereas a set of received input symbols is the right choice of vertices for the transition process dependence graph, capturing the adjacency relation of the monitor dependence graph is most straightforward when each vertex is chosen to be a list of the form

$$\langle d, b, w, g \rangle \in \mathcal{P}(\mathbb{N})^4.$$

Each term is a set of indices of transitions relative to the ordering a with its own particular interpretation.

- d contains the indices of the transitions from which lock requests are deniable because a lock is going to be granted to one of their competitors.
- b contains the indices of the transitions from which lock requests are blockable because a lock request from them has already been denied since their most recent revocation handshake acknowledgment and a grant to a competitor of theirs is still pending. (See page 554.)
- w contains the indices of the transitions from which revocation acknowledgments are awaited.
- g contains the indices of the transitions to which a lock will be granted when all revocation handshakes on their competitors complete.

Building the dependence graph amounts to obtaining the vertices $m \in \mathcal{P}(\mathbb{N})^4$ and corresponding adjacency sets $e \in \mathcal{P}(\mathbb{D}^* \times \mathcal{P}(\mathbb{N})^4)$ in pairs (m, e) as usual, which is solved for the most part by a method of obtaining e from m . For this purpose, the list $\langle c, r, l, h \rangle$ of parameters in Equation 16.37 determines a second order function

$$\vec{j}\langle c, r, l, h \rangle : \mathcal{P}(\mathbb{N})^4 \rightarrow \mathcal{P}(\mathbb{D}^* \times \mathcal{P}(\mathbb{N})^4)$$



such that $\vec{j}\langle c, r, l, h \rangle m = e$ yields the desired adjacency set as a union of four subsets

$$\vec{j}\langle c, r, l, h \rangle m = \bigcup_{k=0}^3 \vec{j}_k\langle c, r, l, h \rangle m$$

when functions \vec{j}_0 through \vec{j}_3 are defined as the foregoing preparation inevitably obliges. Any process in the ensemble of mutually dependent processes collectively defining the monitor can accept a lock request signal l_{i0} from the i -th transition provided this transition is not deniable, blockable, grantable, or charged with acknowledging a revocation request. In that event, the process outputs all revocation request signals in r_i by Equation 16.35 and amends its subsequent behavior to deny lock requests from transitions identified in c_i . It also waits for revocation acknowledgments from those transitions and commits to granting a lock to the i -th thereafter.

$$\vec{j}_0 = \lambda\langle c, r, l, h \rangle. \lambda\langle d, b, w, g \rangle. \bigcup_{i \in \mathcal{D}(r) - (d \cup b \cup w \cup g)} \{ \langle \langle \text{get } l_{i0}, (\mathcal{F} \text{ par}) \text{put}^*(r_i)^{\circ-1} \rangle, \langle d \cup c_i, b, w \cup c_i, g \cup \{i\} \rangle \rangle \}$$

Any process receiving a lock request l_{i0} from a deniable transition numbered i denies the request by transmitting l_{i2} and behaves similarly thereafter except to the extent of treating lock requests from the i -th transition as blockable rather than deniable.

$$\vec{j}_1 = \lambda\langle c, r, l, h \rangle. \lambda\langle d, b, w, g \rangle. \bigcup_{i \in d} \{ \langle \langle \text{get } l_{i0}, \text{put } l_{i2} \rangle, \langle d - \{i\}, b \cup \{i\}, w, g \rangle \rangle \}$$

A process may also accept a revocation acknowledgment signal h_i from the i -th transition provided it has been waiting for such a signal, after which it treats lock requests from the i -th transition as blockable in this case as well and treats the revocation handshake as having concluded.

$$\vec{j}_2 = \lambda\langle c, r, l, h \rangle. \lambda\langle d, b, w, g \rangle. \bigcup_{i \in w} \{ \langle \langle \text{get } h_i \rangle, \langle d - \{i\}, b \cup \{i\}, w - \{i\}, g \rangle \rangle \}$$

Additionally, a process can grant a lock to i -th transition by concluding a 4Φ handshake over the i -th lock channel provided this transition is grantable and no revocation acknowledgments are awaited from any of its competitors. Thereafter, the process can treat the i -th transition as no longer grantable and its competitors as neither deniable nor blockable.

$$\vec{j}_3 = \lambda\langle c, r, l, h \rangle. \lambda\langle d, b, w, g \rangle. \bigcup_{i \in g \cap (\mu c^{-1}) (\mathcal{R}(c) \cap \mathcal{P}(w - \bigcup \mathcal{R}(c)))} \{ \langle \langle \text{put } l_{i1}, \text{get } l_{i0}, \text{put } l_{i1} \rangle, \langle d - c_i, b - c_i, w, g - \{i\} \rangle \rangle \}$$

Consequently, we have the whole dependence graph in the percolation

$$(\lambda q. (\rho \lambda(m, e). \bigcup_{n \in \mathcal{R}(e)} q n) q \emptyset^4) \lambda v. \{ (v, \bigcup_{k=0}^3 \vec{j}_k\langle c, r, l, h \rangle v) \}$$

to vertices reachable from the vertex $\emptyset^4 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ corresponding to the process with d , b , w , and g identically empty.

Environment

A parsimonious description of the monitor process $(g_4 c) g_3 g_2 s \in \mathbb{D}$ as a fixed point of the system induced by the dependence graph is obtained by [Equation 16.36](#), [Equation 16.38](#), and g_4 defined as

$$g_4 = \lambda c. \lambda f. \Upsilon (\lambda q. (\rho \lambda (m, e). \bigcup_{n \in \mathcal{R}(e)} q n) q \emptyset^{\perp}) \lambda v. \{(v, \bigcup_{k=0}^3 \vec{j}_k(c : f) v)\}$$

but this result is only an open Petri net modeled process by [Equation 16.5](#). As noted on page 557, a closed Petri net modeled process would be more efficient for state based synthesis, and can be obtained by combining this process with a compatible environment.

Fortunately, a compatible environment for the monitor process is much simpler to describe than the monitor process itself. In terms of some of the same parameters $\langle r, l, h \rangle = g_3 g_2 s$ used to specify the monitor process, one aspect of the environment is an incessant clamor of concurrent lock requests on every lock negotiation channel

$$(\mathcal{F} \text{ par}) (\lambda i. \text{loop seq} (\text{put } l_{i_0}, \text{alt} (\text{get } l_{i_1}, \text{get } l_{i_2})))^* \iota_{|l|}$$

and the other is an echo chamber of acknowledgments to any revocation request

$$(\mathcal{F} \text{ par}) (\lambda j. \text{loop seq} ((\mathcal{F} \text{ alt}) (\lambda k. \text{get } r_{jk})^* \iota_{|r_j|}, \text{put } h_j))^* \iota_{|h|}$$

which take place concurrently with each other and can be summarized as $g_5 \langle r, l, h \rangle$ by

$$g_5 = \lambda \langle r, l, h \rangle. \text{par} (\\ (\mathcal{F} \text{ par}) (\lambda i. \text{loop seq} (\text{put } l_{i_0}, \text{alt} (\text{get } l_{i_1}, \text{get } l_{i_2})))^* \iota_{|l|}, \\ (\mathcal{F} \text{ par}) (\lambda j. \text{loop seq} ((\mathcal{F} \text{ alt}) (\lambda k. \text{get } r_{jk})^* \iota_{|r_j|}, \text{put } h_j))^* \iota_{|h|})$$

enabling a behaviorally equivalent closed Petri net modeled monitor process as

$$(\mathcal{F} \text{ env}) (\langle g_4 c, g_5 \rangle^{\Delta} (g_3 g_2 s)^{\perp}).$$

It is only a short step further to the monitor block construction $\text{MON}(a, p) \in \mathbb{H}$ by state based synthesis in terms of the originally stipulated parameters $(a, p) = \dot{U}_z^{\alpha}(u, A)$, [Equation 16.33](#), and [Equation 16.34](#).

$$\text{MON} = (\lambda \langle c, s \rangle. \text{SBS}(g_1 \langle c, s \rangle, (\mathcal{F} \text{ env}) (\langle g_4 c, g_5 \rangle^{\Delta} (g_3 g_2 s)^{\perp}))) \circ g_0 \quad (16.39)$$

16.6.5 Direct mapped transition arrays

The payoff for developing the lockable transition array in [Section 16.6.3](#) and the monitor block in [Section 16.6.4](#) is that a transition array $\text{TRA}(v, a, p) \in \mathbb{H}$ made from their combination in this section may be a feasible alternative to $\text{TRA}_0(v, a, p)$ as developed in [Section 16.5.2](#) when state explosion precludes the latter. With the heavy lifting completed above, we can dispense briefly with the rest as follows.

Monitored transitions

A lockable transition array and a monitor both parameterized by the same terms (a, p)

$$\langle \text{LTR, MON} \rangle^\Delta (a, p)^\Delta$$

are combined partly by having the $|a|$ lock request outputs and $|a|$ revocation acknowledgment outputs from the former connected to the $2|a|$ inputs similarly designated on the latter in a block

$$\mathbf{L}_{2|a|}(\langle \text{LTR, MON} \rangle^\Delta (a, p)^\Delta).$$

The remaining exposed $2|a|$ lock responses and $|\mathring{b} s| = |\mathring{b} g_0(a, p)_1|$ revocation requests by [Equation 16.33](#) can be connected from the outputs to the inputs through a feedback bus of total width

$$b = 2|a| + |\mathring{b} g_0(a, p)_1|$$

by rolling down the $|v|$ token notification inputs and rolling up the $|a|$ firing outputs on a block

$$\mathbf{Z}^b(\mathbf{Z}^b \mathbf{R}(\mathbf{L}_{2|a|}(\langle \text{LTR, MON} \rangle^\Delta (a, p)^\Delta) \downarrow |v| \uparrow |a|, \mathbf{I}^b) \downarrow b)$$

for $v = \bigcup \mathcal{R}(p^* a)$ denoting collective preset places as in [Equation 16.14](#) but treated here as an independent parameter in an expression

$$((g_6 g_0) \langle \text{LTR, MON} \rangle) (v, a, p)$$

for the whole block based on

$$g_6 = \lambda f. \lambda l. \lambda(v, a, p). (\lambda b. \mathbf{Z}^b(\mathbf{Z}^b \mathbf{R}(\mathbf{L}_{2|a|}(l \Delta (a, p)^\Delta) \downarrow |v| \uparrow |a|, \mathbf{I}^b) \downarrow b)) 2|a| + |\mathring{b} (f(a, p))_1|.$$

Isolated transitions

A couple of edge cases affect this result. If there is only one transition in the community, then the monitor is unnecessary and technically undefined, but the transition is adequately simulated by a JOIN network connecting $|v|$ token notification inputs to the single firing output.

$$\langle \langle (g_6 g_0) \langle \text{LTR, MON} \rangle \rangle (v, a, p), \text{JOIN } |v| \rangle_{\delta_1^{|a|}}$$

If there is a lack of preset places, which would imply a community of just one transition, which is an open input, then even the JOIN network is undefined, but the result devolves to a wire following the discussion on page 546.

$$\langle \langle \langle (g_6 g_0) \langle \text{LTR, MON} \rangle \rangle (v, a, p), \text{JOIN } |v| \rangle_{\delta_1^{|a|}, \mathbf{I}} \rangle_{\delta_v^\emptyset}$$

Best efforts

A definition for a transition array favoring the state based synthetic form where feasible can be given in terms of a freely chosen metric $\|\cdot\|$ and constant K_r determined by the available computing resource budget. The general transition array $\text{TRA}(v, a, p) \in \mathbb{H}$ by a function

$$\text{TRA} : \mathcal{P}(\mathbb{V}) \times (\mathbb{T} \cup \mathbb{V})^* \times ((\mathbb{T} \cup \mathbb{V}) \rightarrow \mathcal{P}(\mathbb{T} \cup \mathbb{V})) \rightarrow \mathbb{H}$$

reverts to the state based transition array for values of $\|(a, p)\|$ less than K_r and the direct mapped transition array otherwise.

$$\text{TRA}(v, a, p) = \begin{cases} \text{TRA}_0(v, a, p) & \text{if } \|(a, p)\| \leq K_r \\ \langle \langle \langle (g_6 g_0) \langle \text{LTR, MON} \rangle \rangle (v, a, p), \text{JOIN } |v| \rangle_{\delta_1^{|a|}, \mathbf{I}} \rangle_{\delta_v^\emptyset} & \text{otherwise} \end{cases} \quad (16.40)$$

16.6.6 Communities

As a happy consequence of behavioral equivalence between the state based and direct mapped transition arrays, a definition of direct mapped synthetic communities

$$\text{COM}(\alpha, A, M) = \lambda u. (\lambda(a, p). (\lambda v. \mathbf{C}_{|v|+\delta_{\varnothing}^v} \langle \text{PL}(M, a, p), \text{TRA}(v, a, p) \rangle) \cup \mathcal{R}(p^* a)) \dot{\mathcal{U}}_z^\alpha(u, A) \quad (16.41)$$

can be boringly similar to that of state based synthetic communities by COM_0 in Section 16.5.3. Only the substitution of TRA by Equation 16.40 for TRA_0 by Equation 16.16 distinguishes them. In addition, the same combining form Ω_z^α defined in Section 16.5.4 is useful for direct mapped synthetic communities and enables a more general definition



$$\text{DMS}_1(\alpha, X) = (\lambda(I, O, (P, T, A, M, F)). m_4(\alpha, I, O) \Omega_z^\alpha((T, A), \text{COM}(\alpha, A, M)^* \dot{\mathcal{U}}_z^\alpha(T, A))) \text{RP } X$$

for direct mapping synthesis analogous to DMS_0 by Equation 16.21. If direct mapping synthesis by this method is effective in practice, then maybe at long last there is no need to read further.

16.7 State implosion

On the other hand, while some progress toward mitigating the cost of state enumeration during direct mapping synthesis is achieved by synthesizing each transition separately, the cost is not eliminated. Each transition block according to Section 16.6.2 is obtained by state based synthesis from a process whose state space increases with the transition's preset cardinality and number of competitors. This state space may be small compared to that of the overall specification, but to no avail if it is still infeasibly large. Similar issues could affect the monitor block construction in Section 16.6.4. The rest of this section investigates a solution.

16.7.1 A naive solution

An obvious idea is to deflate the transition and monitor state spaces by direct mapping synthesis of the transition blocks and monitors themselves. For a first attempt, instead of obtaining a state based synthetic transition block $\text{TR}(n)$ as

$$(\lambda \langle i, g, f \rangle. \text{SBS}(g \ i, (\mathcal{F} \ \mathbf{env}) \ f \ i)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle n^3)$$

by Equation 16.28, we could use

$$(\lambda \langle i, g, f \rangle. \text{DMS}_1(g \ i, (\mathcal{F} \ \mathbf{env}) \ f \ i)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle n^3)$$

for a direct mapped synthetic transition block by using DMS_1 in place of SBS as the synthesis method. However, because the current motivation is to avoid enumerating the state space, it would be better to make that

$$(\lambda \langle i, g, f \rangle. \text{DMS}_1(g \ i, (f \ i)_0)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle n^3)$$

using only the open Petri net modeled process $(f \ i)_0 = ((f_5 \ f_3 \ n) \ f_2 \ n)_0$ and omitting the environment, which would otherwise result in a costly conversion to an open Petri net by Equation 16.7. A comparable replacement for the block $\text{MON}(n)$ would be

$$(\lambda \langle c, s \rangle. \text{DMS}_1(g_1 \langle c, s \rangle, (g_4 \ c) (g_3 \ g_2 \ s))) g_0 \ n$$

by an adaptation of Equation 16.39.

16.7.2 A better solution

The solution in [Section 16.7.1](#) might not be completely satisfactory because it still entails state based synthetic transitions and monitor blocks in a less obvious way. The definition of DMS_1 depends indirectly on [Equation 16.28](#) and [Equation 16.39](#) so it inherits their vulnerability to state explosion. A revised attempt calls for a more sophisticated direct mapping synthesis function

$$\text{DMS} : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$$

that does not depend on them.

Recurrence

How should this function be defined? This trouble all started with the observation that not only might the transitions and monitors be too big for state based synthesis, but if their direct mapping synthesis is attempted, then even the transitions and monitors within them might be too big. With no end in sight, the situation appears to call for a recurrence whose base case pertains to specifications that are small enough for state based synthesis, as in

$$\text{DMS}(\alpha, X) = \begin{cases} \text{SBS}(\alpha, X) & \text{if } \|X\| \leq K_s \\ F(\text{DMS}, \alpha, X) & \text{otherwise} \end{cases} \quad (16.42)$$

for a cost measure $\|X\|$ of the candidate process $X \in \mathbb{D}$, an implementation dependent constant K_s marking the threshold of feasibility for state based synthesis, and an inductive case of a form F yet to be established.

Variably synthesized blocks

On another front, if the function DMS envisioned above is not to depend on the forms of transition blocks and monitors defined by [Equation 16.28](#) and [Equation 16.39](#), then what is the alternative? The problem with these methods is that they invoke state based synthesis unconditionally, and modifying them as in [Section 16.7.1](#) for direct mapping synthesis by DMS_1 is not much of an improvement. However, a recursive definition of DMS might allow it to depend on a transition block

$$\widehat{\text{TR}}(\text{DMS}) n \in \mathbb{H}$$

instead of $\text{TR}(n)$ in terms of a higher order function

$$\widehat{\text{TR}} : ((\mathbb{T}^* \times \mathbb{D}) \rightarrow \mathbb{H}) \rightarrow (((\mathbb{T} \cup \mathbb{V}) \rightarrow \mathcal{P}(\mathbb{T} \cup \mathbb{V})) \times \mathcal{P}(\mathbb{T} \cup \mathbb{V}) \times (\mathbb{T} \cup \mathbb{V}) \rightarrow \mathbb{H})$$

defined by

$$\widehat{\text{TR}}(S) = \lambda n. (\lambda \langle i, g, f \rangle. S(g i, (f i)_0)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle n^3)$$

parameterized by a synthesis method $S : \mathbb{T}^* \times \mathbb{D} \rightarrow \mathbb{H}$, or the minor improvement

$$\widehat{\text{TR}}(S) = \lambda n. \begin{cases} \text{TR } n & \text{if } \|n\| \leq K_t \\ (\lambda \langle i, g, f \rangle. S(g i, (f i)_0)) (\langle f_2, f_4, f_5 \circ f_3 \rangle \triangle n^3) & \text{otherwise} \end{cases} \quad (16.43)$$

that generates a more efficient version by not neglecting the environment in the limiting case of measurably low costs $\|n\| \leq K_t$. The analogous remedy for monitor blocks would be

$$\widehat{\text{MON}}(S) = \lambda n. \begin{cases} \text{MON } n & \text{if } \|n\| < K_m \\ (\lambda \langle c, s \rangle. S(g_1 \langle c, s \rangle, (g_4 c) (g_3 g_2 s))) g_0 n & \text{otherwise.} \end{cases}$$

The flexibility inherent in [Equation 16.43](#) is no impediment to a similarly parameterized lockable transition array $\widehat{\text{LTR}}(S)$ $l \in \mathbb{H}$ of a form

$$\widehat{\text{LTR}}(S) = \lambda l. (\lambda \langle h, d, f \rangle. (\mathcal{F} \mathbf{R}) (|d| : (\lambda i. \text{FORK } |(fd)_i|)^* \iota_{|fd|}) \xrightarrow{d \circ \mathfrak{b} f d} h(\widehat{\text{TR}} S, \text{LK})) (\langle f_6, f_7, f_8 \rangle \triangle l^3)$$

inspired by [Equation 16.31](#), leading next to a variably constructed transition array

$$\widehat{\text{TRA}}(S) = \lambda(v, a, p). \begin{cases} \text{TRA}_0(v, a, p) & \text{if } \|(a, p)\| < K_r \\ \langle \langle (g_6 \ g_0) (\widehat{\text{LTR}}, \widehat{\text{MON}}) \triangle S^2 \rangle \rangle (v, a, p), \text{JOIN } |v| \rangle_{\delta_1^{|a|}, \mathbf{1}} \rangle_{\delta_\varnothing} & \text{otherwise} \end{cases}$$

similar to [Equation 16.40](#), and finally a community thereof by

$$\widehat{\text{COM}}(S) = \lambda(\alpha, A, M). \lambda u. (\lambda(a, p). (\lambda v. \mathbf{C}_{|v|+\delta_\varnothing} \langle \text{PL}(M, a, p), (\widehat{\text{TRA}} S) (v, a, p) \rangle) \cup \mathcal{R}(p^* a)) \dot{\mathcal{U}}_z^\alpha(u, A)$$

following [Equation 16.41](#).

Inductive case

We are now well situated to supply the inductive case of the recurrence in [Equation 16.42](#) based on the intervening observations. Taking another cue from [Equation 16.21](#), we can expect something of the form

$$m_4(\alpha, I, O) \Omega_z^\alpha((T, A), ((\widehat{\text{COM}} \text{DMS}) (\alpha, A, M))^* \mathcal{U}_z^\alpha(T, A))$$

where $(I, O, (P, T, A, M, F)) \in \mathbb{D}$ is the refined canonical form $\mathbf{RP} X$ of the synthesis candidate X by [Equation 16.7](#). However, for a less cluttered recurrence, the use of

$$m_5 = \lambda f. \lambda c. \lambda S. \lambda(\alpha, (I, O, (P, T, A, M, F))). f(\alpha, I, O) \Omega_z^\alpha((T, A), ((c S) (\alpha, A, M))^* \mathcal{U}_z^\alpha(T, A))$$

enables a more succinct expression.

$$\text{DMS}(\alpha, X) = \begin{cases} \text{SBS}(\alpha, X) & \text{if } \|X\| \leq K_s \\ (((m_5 \ m_4) \widehat{\text{COM}}) \text{DMS}) (\alpha, \mathbf{RP} X) & \text{otherwise} \end{cases} \quad (16.44)$$

16.7.3 Concluding remarks

On a final note, we might now supplement the concluding remarks in [Section 15.6.2](#) regarding the transformation $\mathcal{J}_{\mathbb{D}\mathbb{H}}^\alpha : \mathbb{D} \rightarrow \mathbb{H}$. Because direct mapping synthesis by [Equation 16.44](#) subsumes state based synthesis, a definition of the transformation by

$$\mathcal{J}_{\mathbb{D}\mathbb{H}}^\alpha(X) = \text{DMS}(\alpha, X)$$

is never worse and sometimes better than a definition by [Equation 15.39](#), and so perhaps on that basis should be preferred.

Reflection hypothesis

1. What happens to the mad scientist simulator if toggle and query inputs are applied concurrently? (hint: It either diverges or arbitrates between them and the answer is not a matter of opinion.)
2. Terminal marked places as in [Figure 16.4](#) convey meaningful information about prohibited input combinations that is lost from the refined canonical form. How could this information be used to enable more efficient monitor blocks in [Section 16.6.4](#) if it were retained?
3. Sending more token notifications to a transition that has not fired since receiving the previous batch smells like a safety violation. How could the edge case discussed on page 554 ever happen without violating the specification (and therefore being fair game for an implementation to ignore)?
4. When, if ever, does the protocol between monitors and transitions allow a monitor to grant two locks concurrently? (hint: [Figure 7.8](#))
5. How hard would it be to use Sperner coded revocation channels between monitors and transition blocks instead of 1-hot channels? (See [Chapter 13](#) and [Chapter 14](#).) What would be the advantages and disadvantages?
6. Draw a more detailed version of [Figure 16.11](#) with every layer of nested parentheses depicted explicitly, and draw similar diagrams for any other block combinator expressions throughout the book whose derivations are not obvious by inspection.
7. A monitor without revocation channels would be called a **nacking arbiter** [133].
 - a) Design a family of nacking arbiters parameterized by the number n of lock negotiation channels in whatever way is easiest.
 - b) Is there any worthwhile way of making a monitor out of a nacking arbiter by glomming something onto it?
8. The **reflection** of a process $X \in \mathbb{D}$ can be defined as the minimum process E in the refinement ordering for which $\mathbf{env}(X, E)$ does not diverge [129, 172].
 - a) Is there an efficient algorithm for computing a function $f : \mathbb{D} \rightarrow \mathbb{D}$ taking a process X to its reflection fX ? (N.B., given X only as a tuple $(I, O, N) \in \mathbb{D}$)
 - b) How could this function be used to solve the so called “design equation” $X = \mathbf{par}(p, Y)$ for an unknown process p given $X, Y \in \mathbb{D}$?
 - c) What implications would an algorithmic solution to the design equation have for the previous question and more generally?



Part V

Appendices

Discretion is knowing how to hide
that which we can not remedy.

Spanish proverb



SUPPLEMENTARY REMARKS ON QUASI-DELAY INSENSITIVITY

The QDI research community can lay claim to a long record of formidable accomplishments, including the world's first fully asynchronous microprocessor [183, 184, 186], and to a scrupulously professed disaffection with delay insensitive design. By way of a recurring theme, DI garners a gracious but firm boilerplate dismissal in the literature as a historical footnote dating from an era before its limitations were understood (as in [16, 34, 64, 75, 131, 176, 203, 230, 231, 233, 267, 268, 279, 281] and countless others). If the much lamented compromise incurred by isochronic forks is the nearest realistic alternative to DI, new initiates can be grateful at least that it is such a small price to pay. Invariably off-message is the idea that a suitably chosen set of primitive components, including some with multiple output terminals, has always enabled general purpose delay insensitive design without further need of isochronic forks [81, 136, 221, 223, 294].

Apologia for this tradition might seek to prevail on a point of principle: most DI primitives amount to no less of a compromise insofar as they conceal internal isochronic forks, these being at least as undesirable. Presumably there is no inherent issue with multiple-output devices *per se*, because it is standard practice in QDI design to use arbitration devices having two outputs [185]. The crux of this position would seem to be that isochronic forks should always be laid bare, even at the cost of additional timing analysis or analog simulation [24], simply as a rejection of undue contrivance and an honest recognition of the hard physical realities of circuit design.

A.1 CMOS inverters

In this case, as noted in [294], the matter illustrated in Figure A.1 poses a dilemma. An inverter is a standard logic gate with a single input and a single output, whose output logic level is the negation or complement of its input. That is, a true input implies a false output and a false input implies a true output, making an inverter just about the simplest logic gate there is. Inverters are

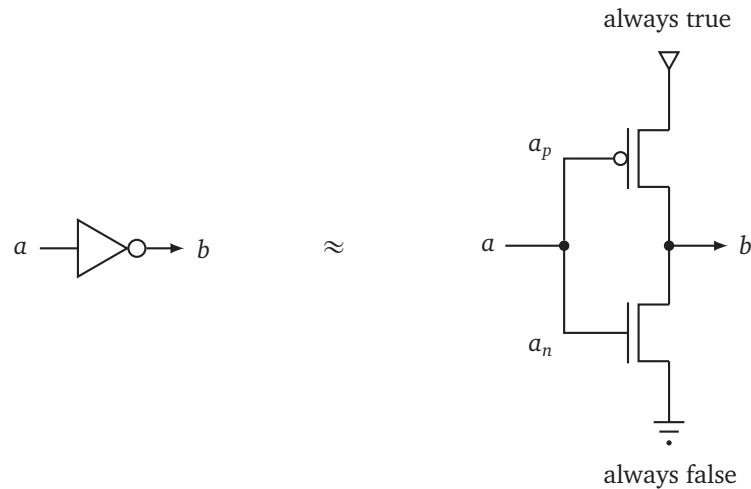


Figure A.1: CMOS transistor-level implementation of an inverter, showing an input a forked to a P-type transistor via branch a_p and to an N-type transistor via branch a_n

ubiquitous in QDI design and have the schematic symbol shown at the left of [Figure A.1](#) (not to be confused with the `MERGE` symbol used in this book). An implementation of an inverter using CMOS technology consists of two complementary transistors connected as shown.

- The transistors act as switches responding in opposite ways (either on or off) to the logic level applied to them by the input a via the fork.
 - A true input on a turns on the lower transistor and turns off the upper one.
 - A false input on a turns on the upper transistor and turns off the lower one.
- The output b sits between two nodes permanently maintained at opposite alternatives (true and false), separated from each of them by one of these switchable transistors, so that the output conveyed to the environment at any given time depends on the states of the transistors as shown in [Figure A.2](#).
 - When the top transistor is on and the bottom is off, the true level at the uppermost node is connected to the output via the top transistor (and the false cut off from it).
 - Alternatively, a false level is communicated to the output through the bottom transistor when that one is on and the top one is off.

A.2 Unexposed delays

The problem with this implementation of an inverter is that the output becomes indeterminate when both transistors are simultaneously off or simultaneously on, even if only for a short time [89]. One word for a transient period of undefined output behavior is “*hazard*”. The simultaneous

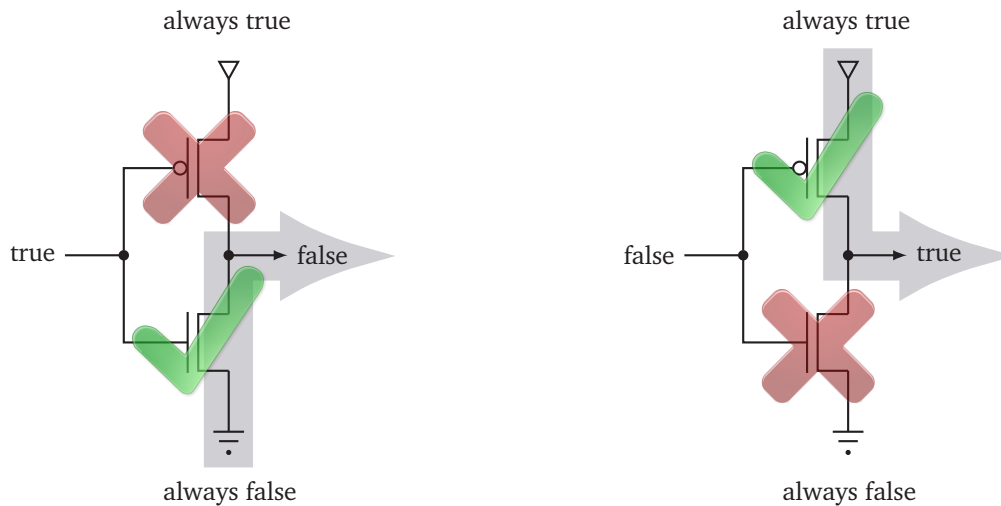


Figure A.2: In the two valid states of a CMOS inverter, the transistors act as switches, with one on and the other off at all times, simultaneously controlled by the input through an isochronic fork.

application of the same input logic level to complementary transistors is meant to prevent hazards by ensuring that the transistors can never be in the same state, but this condition is violated if a change in the level of the input node a propagates at a different rate through the upper branch of the fork a_p than through the lower branch a_n , thereby reaching the transistors at different times, or if one transistor takes longer to switch states than the other.

This problem has a solution. The physical placement and dimensions of the transistors and connecting wires can be chosen to match the delays along each branch of the fork, thus ensuring simultaneous switching between states. In other words, an isochronic fork is needed within the inverter.

A.3 Conclusions

Given that unexposed isochronic forks and multiple-output devices are neither excluded from QDI design in practice nor especially topical in the literature of the subject, a more thorough account of its principles and rationale than what can be surmised from published sources would be preferable for understanding the community's apparent disinterest in DI.

Short of that, some insight is possible by observing the selection of components typically used in QDI design. Traditional logic gates with synchronizing elements and an arbitration device as a concession to asynchrony suggest more of an evolutionary progression from earlier schools of thought than a clear cut paradigm shift [150]. The comfortable familiarity of most of these devices would also be a more plausible explanation for the historical ascendancy of QDI over DI design than the customary citation of [182]. Without the requisite cultural frame of reference and pedagogical commentary, there would be nothing to stop an impartial investigator from interpreting [182] simply as a statement of the limitations of circuits made from forks and logic gates.

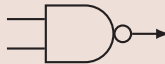
In reality, unless some physical law precludes the use of transistors in anything other than flip flops and Boolean combinatorial networks, only the force of habit restricts anyone to them as a basic abstraction layer for asynchronous system development, and the proper setting for a discussion of genuine engineering compromises is among the possible alternatives. Like any other design decision, this choice confers advantages when made objectively and judiciously, but uncertain returns when misapprehended.

Treasure hunt

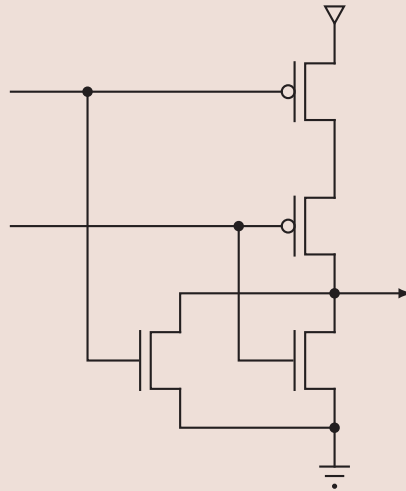
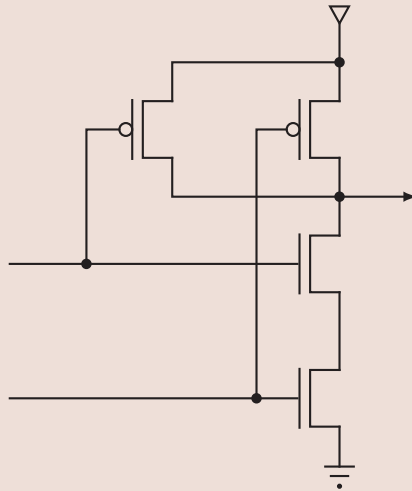
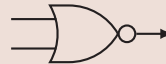
Spot the hidden isochronic forks in these popular QDI logic gates. (hint: Which forks could cause hazards if both transistors controlled by them were to conduct simultaneously?)



NAND



NOR



In making theories, always keep a window open so that you can throw one out if necessary.

Bela Lugosi

A P P E N D I X



COMPLETE PARTIAL ORDERINGS AND FIXED POINTS

The pseudo-fixed point combinator described in [Section 5.4.6](#) is informed by a well known body of theory originating from the study of programming languages. In particular, the concept of recursively defined processes as minimal solutions to functional equations descends directly from the ideas of denotational semantics [\[242\]](#). As compelling and profound as these ideas may be, they are not always as amenable to a treatment of DI processes as one might like. Settling for a pseudo-fixed point rather than actually a fixed point has been only the most noticeable of several expedients.

Perhaps a mathematically talented reader will surpass the author's limited attempt hitherto at a smooth synthesis of formal pedigree and convenience. The purpose of this appendix is to commend to the attention of anyone so inclined a few general points of common ground and of disparity between the classical development of recursion via fixed points and the way it is handled in this book. We start with a brief overview of the basic standard concepts and proceed from there.

B.1 Theoretical primer

The standard approach to establishing the existence of a fixed point starts by supposing that functions whose fixed points are of interest operate on data values in a **complete partial ordering** [\[102, 240, 270\]](#).¹ A complete partial ordering, or CPO, is defined as a set D with these additional features.

- a transitive, reflexive, antisymmetric (*i.e.*, partial order) relational operator denoted \sqsubseteq
- a unique minimum element $\perp \in D$ (read “bottom”), which is to say $\forall x \in D. \perp \sqsubseteq x$
- a **least upper bound** in D , denoted $\bigsqcup S$, for any **directed set** $S \subseteq D$

¹Lattices rather than CPOs are used in [\[270\]](#), but much of the same discussion applies. See [\[2\]](#) for a broader view.

The least upper bound of a set $S \subseteq D$ is the minimum $x \in D$ satisfying $\forall a \in S. a \sqsubseteq x$, which is the minimum member of D that dominates every member of the set S . A directed set S satisfies $\forall a, b \in S. \exists c \in S. a \sqsubseteq c \wedge b \sqsubseteq c$, which means any two members of the set have a common member above both of them.

Finite directed sets always have a least upper bound (namely their maximum element), but “completeness” of a CPO demands the same of infinite directed sets, which is no small distinction. By way of analogy, the set of rational numbers less than $\sqrt{2}$ under the usual ordering has a least upper bound (namely $\sqrt{2}$) but the least upper bound is not a rational number. Nor is any rational $x < \sqrt{2}$ an adequate substitute, because there is always another one greater than x but less than $\sqrt{2}$. The completeness condition helps to formalize a notion of continuity.

B.1.1 Standard fixed point construction

For a complete partial ordering D , the usual way of defining a fixed point of a function $f : D \rightarrow D$ is as the least upper bound of a series of increasingly well defined approximations of it,

$$\text{fix}(f) = \bigsqcup \{f^i(\perp) \mid i \in \mathbb{N}\} \quad (\text{B.1})$$

where exponentiation of a function with a natural number represents iterated composition as explained in reference to [Equation 6.2](#). The fix function in [Equation B.1](#) is the generic fixed point function for a CPO, not necessarily related to the fix combinator defined in [Equation 5.31](#) except insofar as the current line of inquiry might yet establish.

B.1.2 Continuity

If $\text{fix}(f)$ is to exist as a member of D , then according to [Equation B.1](#) the least upper bound of the infinite set $\{\perp, f(\perp), f(f(\perp)), \dots\}$ must exist in D , which is guaranteed only if this set is directed. If the function f is **monotonic**, which is to say

$$\forall x, y \in D. x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

then clearly the set is directed and the least upper bound exists. For the least upper bound to be a fixed point of f (i.e., a solution to $X = f(X)$) requires in addition that f be **continuous**, which means

$$f(\bigsqcup S) = \bigsqcup \{f(x) \mid x \in S\} \quad (\text{B.2})$$

for any directed $S \subseteq D$.² It can be shown that if a continuous function f has more than one fixed point, then the one obtained from [Equation B.1](#) is the least fixed point with respect to this ordering.

A first encounter with the standard fixed point construction may give the impression that it can be of only limited use because of its restriction to continuous functions, especially to a reader versed in an applied subject where discontinuous functions are commonplace. However, this concept of continuity differs from the interpretation of the term familiar from textbooks. Continuity in the present sense (which implies monotonicity) is a reasonable condition for functions to meet if the \sqsubseteq relation represents some kind of information ordering, with \perp being the least informative value.

For a toy example where continuity is a natural and appropriate condition, suppose functions $f : D \rightarrow D$ represent programs, \perp represents the perpetually postponed result of a non-terminating

²A definition of continuity restricted to **chains** can also work, where a chain S satisfies $\forall a, b \in S. a \sqsubseteq b \vee b \sqsubseteq a$.

program, and D is a **flat CPO** of all possible program inputs and outputs, meaning that all members of D other than \perp are mutually unrelated by \sqsubseteq with only \perp below them. In this model, a discontinuous function would represent a program that fails to terminate for at least one input in D other than \perp , but terminates when the input is \perp . While the former property is meaningful if undesirable, the latter would be unrealistic if running a program were understood to entail waiting for its input to become available. In other settings, continuity can be made to coincide more or less with conventional notions of computability.

B.1.3 Ordering of functions

The \sqsubseteq relation associated with any CPO of a set D extends pointwise to functions operating on it. For functions $g, h : D \rightarrow D$, we write $g \sqsubseteq h$ to express the condition $g(x) \sqsubseteq h(x)$ for all $x \in D$. We restrict attention technically to total functions of D , but the entrenched convention is to describe a function h informally as “undefined” for values of $x \in D$ where $h(x)$ is defined as \perp . This usage can be blamed on the custom of \perp representing undefined behavior of the program, process, circuit, or other entity modeled by the function h in typical applications of this theory.

Although it may seem an abuse of notation to overload the relational operator by writing $g \sqsubseteq h$ in reference to functions operating on D , it is actually quite *apropos* because this relationship meets all of the conditions for a CPO itself, albeit one distinct from D .

- The \sqsubseteq relation on functions of D is reflexive, transitive, and antisymmetric. Check.
- The \perp element of the CPO of functions of D is the function that is undefined (wink, wink) for all $x \in D$.
- A least upper bound $f = \bigsqcup S$ exists for any directed set of functions $S \subset D \rightarrow D$. The value of $f(x)$ for any $x \in D$ is as follows.
 - If $h(x)$ is undefined for all $h \in S$, then $f(x)$ is undefined.
 - If $h_i(x)$ is defined for only a single $h_i \in S$, then $f(x)$ is the value of $h_i(x)$ for the h_i that is defined at x .
 - If multiple members $h \in S$ are defined at x , then $f(x)$ is the least upper bound of their values with respect to the ordering on D , which exists because S is directed.

The last condition, which subsumes the other two, can be stated more formally as

$$\bigsqcup S = \lambda x. \bigsqcup (\mu \lambda h. h x) S$$

where the \bigsqcup operator pertains to the CPO of total functions $D \rightarrow D$ on the left but to the original CPO of data values, D , on the right, and the set mapping operator μ is defined by [Equation 5.1](#).

An intuitive concept of the relational operator \sqsubseteq on functions of a CPO is that it orders them by information content. That is, functions higher up in the ordering either define meaningful results for a larger set of inputs than those below them, or yield more informative results for the same inputs, or both.

B.2 Relevance to DI processes

The theory sketched above appears to call for a CPO of DI processes ordered by refinement as defined in Equation 7.18 with the initially divergent process depicted in Figure 7.13 as the \perp element. In this way, the limit construction in Equation B.1 would imply an elegant alternative analytical form of the **fix** combinator defined by Equation 5.31 and impart to functions f expressed by first order process combinators like **seq**, **par**, *etc.* a refinement ordering of their own. Verification by formal proof techniques such as fixed point induction might then complement known methods of model checking based on trace set containment. Whether this way of thinking would be viable and advantageous depends on how capably the following issues might be resolved.

B.2.1 CPO Structure

Two technical obstacles to the construction of a CPO ordered by refinement from the set of DI processes \mathbb{D} are the antisymmetry condition and the uniqueness of a minimum element. Possible workarounds are discussed below.

Antisymmetry

While relational trace set containment certainly is a partial order relation, refinement as currently defined is not, because the antisymmetry condition fails. Based on Section 7.4.3, refinement between a pair of DI processes $X = (I_X, O_X, N_X)$ and $Y = (I_Y, O_Y, N_Y)$ is defined by

$$X \sqsubseteq Y \Leftrightarrow I_X = I_Y \wedge O_X = O_Y \wedge \llbracket X \rrbracket \supseteq \llbracket Y \rrbracket. \quad (\text{B.3})$$

Antisymmetry would require $X \sqsubseteq Y \wedge Y \sqsubseteq X \Rightarrow X = Y$ to hold, but two processes X and Y could easily refine each other without being equal to each other by having different Petri net models N_X and N_Y that give rise to the same relational trace set $\llbracket X \rrbracket = \llbracket Y \rrbracket$.

As noted in [2], dropping the antisymmetry requirement makes the structure a *preorder* rather than a CPO, but the current situation could be salvaged partly by recasting refinement in terms of a CPO populated by equivalence classes of processes. In this context, it would not be valid to think of one process refining another, but instead to define a refinement relation only on their respective behavioral equivalence classes

$$[X] \sqsubseteq [Y] \quad (\text{B.4})$$

using the conventional equivalence class notation $[X]$ for the set of DI processes

$$[X] = \{Y \in \mathbb{D} \mid I_X = I_Y \wedge O_X = O_Y \wedge \llbracket X \rrbracket = \llbracket Y \rrbracket\}$$

behaviorally equivalent to a process X . The condition expressed by Equation B.4 could then be defined to mean Equation B.3 holds for all $(X', Y') \in [X] \times [Y]$.

Unique minima

Use of equivalence classes does not suffice to make the set of DI processes conform to a CPO structure under the refinement ordering because the requirement of a unique \perp element is still unmet. No class $[(I, O, N)]$ with $\llbracket (I, O, N) \rrbracket = (I \cup O)^*$ can have any proper predecessors under the ordering of Equation B.4, and there is a distinct class of processes having this property for every union of non-intersecting alphabets $I, O \in \mathcal{P}(\mathbb{T})$.

- One way of resolving this matter is to restrict attention to a CPO of classes of processes having alphabets I and O in common.
- Another is to think big by having the universal CPO of all equivalence classes in $\mathcal{P}(\mathbb{D})$ ordered with respect to alphabet containment as a priority and relational trace set containment only among classes with compatible alphabets.

The latter remedy implies the universal minimum element

$$[(\emptyset, \emptyset, (\{p, q\}, \{r\}, \{(p, r), (r, q)\}, \{p, q\}, \emptyset))]$$

for arbitrary distinct $p, q, r \in \mathbb{V}$. (See [Figure 7.13](#) and the related discussion.)

B.2.2 Least upper bounds

Because the motivation for this exercise is an alternative route to a fixed point combinator for DI processes by way of [Equation B.1](#), and because [Equation B.1](#) defines a fixed point as the least upper bound of an infinite series of approximations, some consideration of the meaning of least upper bounds in this context is necessary.

The least upper bound of a set of equivalence classes $S \subseteq \mathcal{P}((I \times O \times \mathbb{P}) \cap \mathbb{D})$ with alphabets I and O in common follows naturally from a refinement relation on equivalence classes as considered in [Section B.2.1](#). The alphabets of any member of $\bigsqcup S$ are the same I and O as any member of a class in S . The relational trace set common to the members of $\bigsqcup S$ is the cumulative intersection of the relational trace sets of the members of the classes in S .

$$\bigsqcup S = \{Z \in (I \times O \times \mathbb{P}) \cap \mathbb{D} \mid \llbracket Z \rrbracket = \bigcap_{[X] \in S} \llbracket X \rrbracket\} \quad (\text{B.5})$$

The least upper bound is always defined for finite or infinite sets, hence satisfying the completeness condition for a CPO by construction.

Although the concept of least upper bounds proposed in [Equation B.5](#) leads generally to infinite sets and is of no help computationally, it may nevertheless allow some sense to be made of an expression such as the following,

$$\llbracket X \rrbracket = \bigsqcup \{(\mu \lambda x. \text{seq}(\text{seq}(\text{get } a, \text{put } b), x))^i \perp \mid i \in \mathbb{N}\} \quad (\text{B.6})$$

which is none other than the least fixed point $\text{fix}(f)$ of a function

$$f = \mu \lambda x. \text{seq}(\text{seq}(\text{get } a, \text{put } b), x) \quad (\text{B.7})$$

by [Equation B.1](#), where the μ operator is necessary to map the process combinators over an equivalence class of processes. (See [Equation 5.1](#).) In particular, technical difficulties pertaining to infinite Petri nets are avoided because every member of every class $f^i(\perp)$ is finite even if the class itself is infinite. [Equation B.5](#) insists only that the limit of this sequence refine all of them, not that it involve any manner of impossible concrete representation.

More good news is that there may be reasons to think the least fixed point $\llbracket X \rrbracket$ specified in [Equation B.6](#) is something we already know how to compute by other means. For the function f in [Equation B.7](#) and $n \in \mathbb{N}$, the class of processes $f^n(\perp)$ contains those that engage in exactly n two-phase handshakes with their environment (assuming the environment co-operates) and then

diverge. A process class that refines $f^n(\perp)$ for any n would have to contain processes capable of infinitely many handshakes without ever diverging. Such a thing can be built to order by the **fix** combinator.

$$\begin{aligned} [X] &= \text{fix } \mu \lambda x. \text{seq} (\text{seq} (\text{get } a, \text{put } b), x) \\ &= [\text{fix } \lambda x. \text{seq} (\text{seq} (\text{get } a, \text{put } b), x)] \end{aligned}$$

The validity of this claim would depend on continuity conditions to be explored briefly in [Section B.2.3](#). If justified, then it enables some new flexibility in reasoning about the pseudo-fixed point combinator within the framework of a CPO, such as manipulations based on the following.

$$\begin{aligned} \text{fix } \mu h &= [\text{fix } h] \\ &= (\lambda x. [x]) \text{fix } h \\ \text{fix } \circ \mu &= (\lambda x. [x]) \circ \text{fix} \end{aligned}$$

B.2.3 Continuity of process combinators

One is entitled to infer $[X] = f[X]$ from $[X] = \bigsqcup \{f^i(\perp) \mid i \in \mathbb{N}\}$ only if f is a continuous function of process equivalence classes. If a function f such as that of [Equation B.7](#) does not satisfy [Equation B.2](#), then this theory as it stands is not useful for reasoning about DI processes. Some provision should be made therefore to establish that functions of interest are continuous.

To assess what continuity might mean in the current setting, a specific example of an infinite directed set may be helpful. A hierarchy of process classes induced by the function f defined in [Equation B.7](#)

$$[X_i] = f^i(\perp)$$

forms an ascending chain under the refinement ordering based on the idea that each member of $[X_i]$ participates in one more handshake with its environment than any member of $[X_{i-1}]$, but one less than any member of $[X_{i+1}]$ before diverging. Its least upper bound can be denoted

$$[X_\omega] = \bigsqcup \{[X_i] \mid i \in \mathbb{N}\}.$$

Because $\{[X_i] \mid i \in \mathbb{N}\}$ is a directed set with a least upper bound of $[X_\omega]$, any continuous function g must satisfy

$$g[X_\omega] = \bigsqcup \{g[X_i] \mid i \in \mathbb{N}\}. \quad (\text{B.8})$$

A couple of routes toward establishing continuity of process combinators are sketched below using this example.

The low road

There is at least one process-combinator expressible function for which an intuitive argument can be made in favor of satisfying [Equation B.8](#), namely f itself. Preceding each $[X_i]$ with an additional handshake, as f does, transforms the set of $\{[X_i] \mid i \in \mathbb{N}\}$ to $\{[X_{i+1}] \mid i \in \mathbb{N}\}$, which should not affect its least upper bound. Preceding an infinite sequence of handshakes with another one produces the same infinite sequence, so $f[X_\omega] = [X_\omega]$ holds as it should if the class of $[X_\omega]$ is a fixed point of f .

To take another example, let g be defined as $\mu \lambda x. \text{par}(K, x)$ for some constant process K . Then [Equation B.8](#) becomes

$$[\text{par}(K, X_\omega)] = \bigsqcup \{[\text{par}(K, X_i)] \mid i \in \mathbb{N}\} \quad (\text{B.9})$$

at least if it is safe to assume $x \equiv y \Rightarrow \text{par}(K, x) \equiv \text{par}(K, y)$. It is not impossible to believe that putting K in parallel with each X_i in the set would manifest itself as if K were in parallel with their least upper bound. For any non-quiescent or divergent K with an output alphabet $\{b\}$, both sides devolve to \perp . In the alternative of a deadlocked K sharing the input a with X_i or X_ω , both sides inherit the misfeature of being able to deadlock non-deterministically. In the case of a deadlocked K with no input (*i.e.*, a “do nothing” process), g has no effect on the behavior of either side of the equation, and so on for other conceivable cases of K .

Would this lucky streak hold out for all functions of interest and all directed sets? The answer depends partly on what functions are of interest. Barring any unforeseen difficulties, a good choice would be the set of all functions expressible by process combinators. It would be straightforward to give an inductive definition of this set. For a sufficiently skilled and motivated reader, a formal proof of continuity by structural induction on this definition might be forthcoming.

The high road

A more ambitious but less certain course is to seek the “book proof” [114] of continuity of process combinators, which might take the form of a contradiction following from the hypothesis of a discontinuous function expressed thereby, or perhaps a subtle countability argument. A discontinuous function g would have to violate Equation B.8 or something similar to it by mapping $[X_\omega]$ to a result that is either unrelated or below $g[X_i]$ in the refinement ordering for some $i \in \mathbb{N}$.

The qualitative difference between something like $[X_\omega]$, the limit of an infinite ascending chain, and $[X_i]$, a member of the chain, is captured formally by describing the latter as **compact**. Following [102], a member x of a CPO is compact if any directed set M with $x \sqsubseteq \bigsqcup M$ contains a member y such that $x \sqsubseteq y$ holds. The concept of compactness is relevant to any CPO, and can often serve as a proxy for the intuitive idea of being finitely describable. For example, rational numbers are compact members of the CPO of real numbers under the usual ordering.

To construct a discontinuous function g so as not to satisfy Equation B.8, this definition would do the trick.

$$g(x) = \begin{cases} x & \text{if } x \text{ is compact} \\ \perp & \text{otherwise} \end{cases} \quad (\text{B.10})$$

We would like to be able to claim that this function is so pathological that nothing like it could ever crop up in practice, so there is no need to worry about discontinuous functions coming from process combinators.

One route to this result would be to establish that a function like Equation B.10 is not computable, whereas anything of practical interest obviously is. The computability of this function depends on the existence of a decision procedure to recognize compact classes of DI processes. Unfortunately, this possibility can not be ruled out immediately. Perhaps detecting cycles in their Petri net models, which could have been put there only by the **fix** combinator, would be effective. However, establishing the inexpressibility of any such algorithm by process combinators is a lower standard to meet.

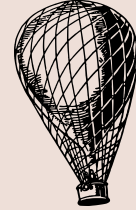
B.3 Further work

Despite the possibilities mentioned at the end of Section B.2.2, it is still far from clear that the classical treatment of fixed points has anything more to offer the DI circuit designer than a Procrustean bed. Although this thesis or its refutation would be *de rigueur* in an academic publication, a conclusive ruling on the issue is not essential to the current text, nor properly within the author’s areas of

interest or expertise. Moreover, it is hoped that by leaving something on the table, this brief introduction to fixed points and complete partial orderings may inspire some industrious reader to do something original with them.

Fixer uppers

1. Let $\hat{\mathbb{D}} \subset \mathbb{D}$ represent the set of DI processes expressible in terms of process combinators.
 - a) Give a formal inductive definition of $\hat{\mathbb{D}}$.
 - b) Assuming the universe \mathbb{T} of symbols in the alphabets of members of \mathbb{D} is countable, prove that $\hat{\mathbb{D}}$ is countable by constructing a bijective function $f : \hat{\mathbb{D}} \rightarrow \mathbb{N}$.
2. Call a function $f : \mathbb{D} \rightarrow \mathbb{D}$ “equivalence preserving” if $x \equiv y$ implies $f(x) \equiv f(y)$ for all $x, y \in \mathbb{D}$. Which of these functions are not equivalence preserving, and why not?
 - a) $\lambda x. \text{par}(K, x)$
 - b) $\lambda x. \text{alt}(K, x)$
 - c) $\lambda x. \text{seq}(K, x)$
 - d) $\lambda x. \text{seq}(x, K)$
3. Would it make any difference in the previous question to restrict the definition of equivalence preservation to expressible processes $x, y \in \hat{\mathbb{D}}$ as defined in the first question?
4. What implication would a lack of equivalence preservation have for continuity of process combinators on a CPO whose members are behavioral equivalence classes of processes?
5. If equivalence preservation is not enforced, what slightly stronger equivalence relation than behavioral equivalence can be defined as a workaround that would be preserved by all functions in Question 2? (hint: Review the discussion leading up to the **fix** combinator in [Section 5.4.6](#).) What is the corresponding refinement relation?
6. Would it be possible to dispense with equivalence classes by restricting attention to processes in canonical form per [Section 7.5.3](#)? If so, would there be any downside?



The deepest sin against the human mind is to believe things without evidence.

Thomas H. Huxley



DECISION WAIT METRICS

Chapter 10 alludes to the optimization of decision wait decomposition strategies with respect to arbitrarily chosen cost or performance metrics but is short on specifics. The only example given is the metric $\|z\| = |\mathcal{F}_{\mathbb{H}\mathbb{L}} z|$, the total number of primitive components in a decision wait $z \in \mathbb{H}$ as determined by the length of the netlist $\mathcal{F}_{\mathbb{H}\mathbb{L}} z \in \mathbb{L}$ in terms of Equation 8.23. This example is not especially interesting in itself because it is intuitively clear that the lowest component count for a planar decision wait is always procured by the cascading decomposition, and for multidimensional decision waits, a straightforward analysis confirms that the crossbar can always beat the dendriform decomposition on component count, even if only by a constant factor. A more realistic requirement is to find the best performance subject to a fixed maximum cost constraint, or to find the lowest cost for a minimum performance constraint, or to optimize a weighted combination of cost and performance.

Another issue is the difficulty of searching an astronomically large design space efficiently or even selecting a representative sample. General purpose techniques for combinatorial optimization are beyond the scope of this book¹, but narrowing the search within the space of decompositions $t \in \mathbb{S}$ as far as possible before generating the decision waits $\hat{\Omega}_g t \in \mathbb{H}$ on which to evaluate the metric

$$\|\hat{\Omega}_g t\| \in \mathbb{R}$$

can only be helpful. The component count metric is a prime example of one suitable to infer directly from the decomposition t by simple arithmetic, as is the critical path length, a rough proxy for latency that ignores wire propagation delays and technology dependent factors in exchange for being possible to calculate without running a routing algorithm or solving physics problems.

This appendix focuses on the computation of the two metrics noted above directly from the decompositions, and even then in a somewhat idealized form. It is intended nevertheless to suggest

¹See [301] for a thorough introduction and reference.

a starting point to practitioners interested in deriving more sophisticated cost and performance criteria suitable for their requirements.

C.1 Component count

The total number of primitive components needed for a decision wait is a measure of its cost that could be made more informative by assessing a specific cost to each type of primitive. Maybe a MERGE takes more area on a chip than a FORK, but not as much as a JOIN, and their costs measured in square nanometers should be tallied accordingly. This cost estimate would still neglect that of the interconnecting network, but would be not much more difficult to compute than the component count alone. This point is noted only in passing because taking it into account would complicate the presentation to follow without offering any further insight about the basic methodology.

C.1.1 Multidimensional

The component count metric follows from assessing a dimensionless unit cost to every component. For a decomposition $t \in \mathbb{S}$ giving rise to a decision wait with dimensions $s = (\psi \Delta_g) t \in \mathbb{N}^*$ by Equation 10.25 and Equation 10.35, the component count $M_g t \in \mathbb{N}$ is obtained by a function $M_g : \mathbb{S} \rightarrow \mathbb{N}$ defined in three cases as follows.

$$M_g = \Lambda \lambda((p, d), m). \begin{cases} \sum m & \text{if } |m| = |d| + 1 \\ (|d_1| - 1)(\sum d_0) + \sum m & \text{if } |m| = |d_1| + \prod d_0 \\ \langle M_q(d, m), M_p \langle d_{00}, d_{10} \rangle \rangle_{\delta_e^m} & \text{otherwise} \end{cases} \quad (\text{C.1})$$

Because M_g is defined as a fold over its argument, the term $(p, d) \in \mathbb{N}^{**} \times \mathbb{N}^{**}$ can be viewed as a node in a decomposition $t = ((p, d), v) \in \mathbb{S}$, while the term $m \in \mathbb{N}^*$ is viewed as the list of the costs of the subtrees already visited (Section 10.1). The permutations p do not appear anywhere else because they do not affect the cost of the components.

The type of decomposition can be inferred from the number of subtrees and hence from the length of m in Equation C.1. The first case pertains to a dendriform decomposition, whose cost is only the sum of the costs of the subtrees because according to Figure 10.16 and Equation 10.20, combining them requires no additional components. The second case pertains to the crossbar decomposition, whose cost is the sum of the costs due to the subtrees in addition to that of the FORK network shown in Figure 10.17 and Equation 10.22. Each network FORK $|d_1|$ requires $|d_1| - 1$ FORK primitives (cf. Figure 9.16), and there are $\sum d_0$ of them. The last case covers planar decision waits, which could be either quadrangular or cascading. Their costs are measured respectively by functions $M_q : \mathbb{N}^{*2} \times \mathbb{N}^* \rightarrow \mathbb{N}$ and $M_p : \mathbb{N}^2 \rightarrow \mathbb{N}$, the latter being relevant to terminal decompositions.

C.1.2 Quadrangular

A cost metric for quadrangular decision waits with dimensions $d \in \mathbb{N}^{*2}$ and building blocks with costs $m \in \mathbb{N}^*$ is given by $M_q(d, m)$ as defined below.

$$M_q(d, m) = \begin{cases} m_2 & \text{if } |d_0||d_1| = 1 \\ m_1 + \sum (\lambda_i \cdot (1 - \delta_1^{d_{0i}})(2d_{0i} + m_{i+2} - 1))^* \iota_{|d_0|} & \text{if } |d_0| \neq 1 \wedge |d_1| = 1 \\ m_0 + \sum (\lambda_i \cdot (1 - \delta_1^{d_{1i}})(2d_{1i} + m_{i+2} - 1))^* \iota_{|d_1|} & \text{if } |d_0| = 1 \wedge |d_1| \neq 1 \\ (2 \sum d) - |d| + \sum m & \text{otherwise} \end{cases} \quad (\text{C.2})$$

This definition adheres closely to [Equation 10.18](#), which specifies a quadrangular decision wait $\Omega_q(d, x, y) \in \mathbb{H}$ in terms of building blocks $x \in \mathbb{H}^2$ and $y \in \mathbb{H}^{|d_0||d_1|}$, such that m_0 is the cost of x_0 , m_1 is the cost of x_1 , and m_{i+2} is the cost of y_i for $0 \leq i < |y|$. These conditions follow naturally in the context of [Equation C.1](#).

- The degenerate case of $|d_0||d_1| = 1$ corresponds to the whole decision wait reducing to y_0 with a cost of m_2 .
- The case of $|d_0|$ and $|d_1|$ both greater than 1 assesses a cost equal to that of the building blocks in addition to that of the completion detecting buses shown in [Figure 10.9](#). A typical completion detecting bus corresponding to a dimension $b \in \mathcal{R}(b d)$ consists of b FORK primitives and $b - 1$ MERGE primitives, hence $2b - 1$ primitives of either type and a total of $(2 \sum b d) - |b d|$ over all dimensions.
- The other two cases pertain to vertical or horizontal quadrangular decision waits. The vertical quadrangular decision wait cost due to the building blocks includes only the cost m_1 of the column input routing stage x_1 and the cost m_{i+2} for blocks y_i having d_{0i} rows with $d_{0i} > 1$. The cost due to the completion detecting buses is $2d_{0i} - 1$ also limited to dimensions d_{0i} greater than 1. The cost of a horizontal quadrangular decision wait is obtained similarly.

C.1.3 Cascading

An expression for the cost $M_p(s) \in \mathbb{N}$ of a cascading decision wait with dimensions $s \in \mathbb{N}^2$ is straightforward to build from expressions $M_l(n)$ and $M_b(n)$ for the respective costs of lateral and bilateral decision waits with n columns.

Lateral decision wait costs

By inspection of [Figure 10.1](#), there is a SHUNT and TOGGLE combination for each of $n - 1$ columns of the lateral decision wait, a MERGE network with n inputs implying $n - 1$ MERGE primitives, and a JOIN, for a total of $2(n - 1) + (n - 1) + 1$ or

$$M_l(n) = 3n - 2$$

primitives of any type.

Bilateral decision wait costs

To follow the pattern of [Figure 10.3](#), the cascading bilateral decision wait contains $2(n - 1)$ bilateral decision wait cells of 5 components each based on [Figure 10.4](#), a MERGE network also with $n - 1$ primitives, and a lateral decision wait with $M_l(2) = 4$ primitives, for a total of $10(n - 1) + (n - 1) + 4$, simplified to

$$M_b(n) = 11n - 7 \tag{C.3}$$

primitives. This result holds even if the number of columns n is equal to 1, when [Equation 10.7](#) simplifies the implementation to that of a rotated lateral decision wait with two columns and a cost of only $M_l(2) = 4$ primitives.

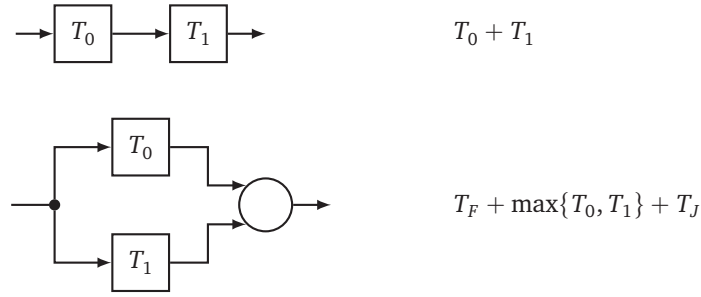


Figure C.1: The critical path length through two blocks in series is the sum of their respective critical path lengths (above), whereas the critical path length through two synchronized blocks in parallel is the maximum thereof (below).

Planar decision wait costs

To follow the pattern of [Figure 10.6](#) a cascading decision wait with s_0 rows and s_1 columns requires $s_0(s_1 - 1)$ planar decision wait cells of 8 primitives each based on [Figure 10.7](#), a MERGE network of $s_1 - 1$ primitives, and a columnar decision wait of $M_l(s_0) = 3s_0 - 2$ primitives, for a total of $8s_0(s_1 - 1) + (s_1 - 1) + 3s_0 - 2$ or more simply

$$8s_0s_1 - 5s_0 + s_1 - 3.$$

This result holds even if both dimensions are unity, in which case by [Equation 10.11](#) the cascade reduces to a lateral decision wait reducing to a single JOIN with a cost $M_l(1) = 1$. However, [Equation 10.11](#) also makes special provisions for dimensions equal to 2 by simplifying the result to a bilateral form. These cases incur a cost of either $M_b(s_0)$ for $s_1 = 2$ or $M_b(s_1)$ for $s_0 = 2$, suggesting a combined result of

$$M_p(s) = \langle\langle 8s_0s_1 - 5s_0 + s_1 - 3, 11s_1 - 7 \rangle_{\delta_2^{s_0}}, 11s_0 - 7 \rangle_{\delta_2^{s_1}}$$

by [Equation C.3](#). This result along with [Equation C.2](#) completes the construction of the cost metric M_g defined in [Equation C.1](#).

C.2 Critical path length

Next on the agenda is the formulation of a performance metric for decision waits based on critical paths. A path can be defined somewhat informally for our purposes as a sequence of components through which a signal propagates to get from an input terminal to a visible output. If there is more than one path for the signal to take concurrently, with a conclusion implied only when all of them have been traversed, then the *critical* path is the longest one (by some definition of “longest”). Concurrent paths exist whenever a signal passes a FORK. Other things being equal, a circuit that does the same job as another but has shorter critical paths should be faster, so critical path lengths are a reasonably plausible performance metric for rough comparisons. For the rest of this discussion, critical path lengths may sometimes be designated more briefly just as critical paths.

A physically realistic approach would identify a specific constant critical path length with each type of component, such as $T_F \in \mathbb{R}$ for a FORK, $T_J \in \mathbb{R}$ for a JOIN, and so on, or even different

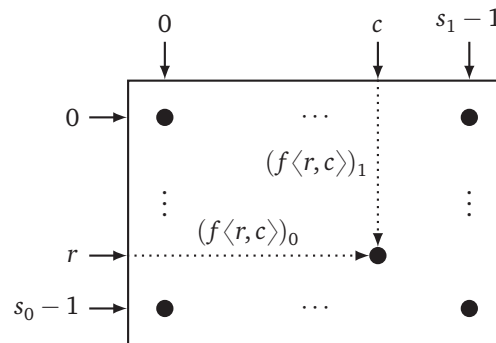


Figure C.2: A planar decision wait with dimensions s_0 -by- s_1 described by a function $f : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ has critical paths from row input r and from column input c to the output in row r and column c listed respectively in the first and second terms of $f\langle r, c \rangle$.

constants for different paths through the same component such as a SHUNT, and then determine the critical paths for their combinations as shown in Figure C.1. However, taking all components to have dimensionless unit critical paths is adequate to demonstrate the method, similarly to the assumption of unit costs in Section C.1.

Less quickly dispatched is the question of how to associate critical paths with circuits having multiple inputs and outputs such as decision waits. A complete description of the critical paths in a decision wait is not just a single number, nor even an assignment of numbers to input terminals, because the critical path from a particular terminal to the output depends on which output it is, which depends on the selection of other inputs. An *ad hoc* solution is to summarize the critical paths in a decision wait with dimensions $s \in \mathbb{N}^*$ by a function

$$f : \mathbb{N}^{|s|} \rightarrow \mathbb{R}^{|s|}$$

such that a list of input terminal numbers $a \in \mathbb{N}^{|s|}$ with $0 \leq a_i < s_i$ determines a list of critical path lengths $f a \in \mathbb{R}^{|s|}$. The value of $(f a)_i$ is interpreted as the critical path from the a_i -th terminal along the i -th axis to the unique output terminal that emits a signal when the a_j -th input on the j -th axis is signaled for all $0 \leq j < |s|$ including i . For example, a row number r and a column number c in a planar decision wait described by a function $f : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ would determine a list of two critical paths $f\langle r, c \rangle \in \mathbb{R}^2$ with $(f\langle r, c \rangle)_0$ being from the r -th row input to the output terminal in row r and column c , and $(f\langle r, c \rangle)_1$ being from the c -th column input the same output terminal as illustrated in Figure C.2.

As a final step, a statistic describing the critical path lengths of a decision wait can be boiled down to a single number for the sake of comparison by averaging or maximizing with respect to f as desired. However, writing a recurrence that expresses the critical paths for a decision wait in terms of those of its building blocks, as we undertake presently, requires access to the full description f pertaining to each building block, so we have to fight the urge to simplify this description prematurely.

One further question to settle before proceeding is that of critical paths through networks of the form FORK n or MERGE n as defined by Equation 9.19 and Equation 9.20. The path is approximately logarithmic in n , but may differ depending on the terminal if n is not a power of two. An exact

result $\tau(n, i) \in \mathbb{R}$ for the i -th terminal is obtained by this recurrence.

$$\tau(n, i) = \begin{cases} 0 & \text{if } n = 1 \\ 1 + \tau(\lfloor n/2 \rfloor, i) & \text{if } i < \lfloor n/2 \rfloor \\ 1 + \tau(\lceil n/2 \rceil, i - \lfloor n/2 \rfloor) & \text{otherwise} \end{cases} \quad (\text{C.4})$$

The rest of this section is devoted to accounting for all critical paths in decision waits starting with cascading forms in [Section C.2.1](#), followed by quadrangular decompositions in [Section C.2.2](#), and finally with multidimensional decompositions in [Section C.2.3](#) through [Section C.2.5](#).

C.2.1 Cascading

The simple case of a lateral decision wait is a good way to start the analysis of cascading decision wait critical paths before progressing to the bilateral and general cases. These critical paths can be reduced to two phases: an *absorption* phase from the input terminal up to the JOIN in [Figure 10.1](#), and an *emission* phase including the JOIN and whatever follows. The total critical path is the sum of these two phases.

Lateral

The absorption phase in a lateral decision wait with n columns is modeled by a function A_l n with the function $A_l : \mathbb{N} \rightarrow (\mathbb{N}^2 \rightarrow \mathbb{R}^2)$ chosen such that $A_l(n) : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ takes row and column numbers $a = \langle a_0, a_1 \rangle \in \mathbb{N}^2$ to a list of two absorption phase critical paths $(A_l n) a \in \mathbb{R}^2$. Because a lateral decision wait can have only one row, $A_l n$ is defined only for row numbers $a_0 = 0$, but the column number a_1 can range from 0 to $n - 1$.

By inspection of [Figure 10.1](#), an input signal to column number a_1 propagates through a SHUNT and a TOGGLE if a_1 is less than $n - 1$, and through a MERGE network regardless before reaching the JOIN. The row input reaches the JOIN immediately and therefore has a zero length absorption phase critical path. These observations translate to

$$A_l(n) = \lambda a. \langle 0, 2(1 - \delta_{n-1}^{a_1}) + \tau(n, a_1) \rangle$$

by [Equation C.4](#). The emission phase is the same for row and column inputs, passing through the JOIN, then through a SHUNT in each of the a_1 columns preceding the input, and then through the SHUNT and the TOGGLE in the input column if it is not the last (*i.e.*, if a_1 is less than $n - 1$), implying an emission phase $E_l(n) : \mathbb{N}^2 \rightarrow \mathbb{R}$ given by

$$E_l(n) = \lambda a. 1 + a_1 + 2(1 - \delta_{n-1}^{a_1})$$

and a total critical path $T_l(n) : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ equal to their sum as noted previously.

$$T_l(n) = \lambda a. (\lambda i. ((A_l n) a)_i + (E_l n) a) * \iota_2 \quad (\text{C.5})$$

Bilateral

Building on this result to that of a bilateral decision wait in a similar style, we seek an absorption phase described by a function $A_b(n) : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ first, with the absorption phase critical paths chosen arbitrarily in this case to include everything from the input terminals through the columnar decision

wait visible in [Figure 10.3](#). The row absorption phase critical path due to a row input $a_0 \in \{0, 1\}$ is already available as $t_1 \in \mathbb{R}$ regardless of the column input, where

$$t = (T_l \ 2) \langle 0, a_0 \rangle \in \mathbb{R}^2$$

is the total critical path associated with column a_0 on a (rotated) two-column lateral decision wait by [Equation C.5](#). In the column absorption phase critical path, there are two bicolunar decision wait cells in any column but the last, interposing two components each to judge by [Figure 10.4](#), or $4(1 - \delta_{n-1}^{a_1})$, followed by a MERGE network with a path of length $\tau(n, n - 1 - a_1)$ borne out by careful reading of [Equation 10.7](#) indicating a reversal of the columns, and then finally t_0 due to the columnar decision wait with t as above, for an overall result

$$A_b(n) = \lambda a. (\lambda t. \langle t_1, 4(1 - \delta_{n-1}^{a_1}) + \tau(n, n - 1 - a_1) + t_0 \rangle) (T_l \ 2) \langle 0, a_0 \rangle.$$

The bilateral emission phase critical path, which is common to both row and column inputs, includes the rest of the way through the bicolunar decision wait cells. Each of the a_1 columns preceding the input column interposes a MERGE and a SHUNT according to [Figure 10.4](#), for a cumulative length of $2a_1$ up to the input column. Subsequently, any column but the last interposes four components from ri to bo for the signal to traverse in one cell, followed by another seven along the path from bi to d in the other, or $11(1 - \delta_{n-1}^{a_1})$ all together, for an emission phase $(E_b \ n) \ a \in \mathbb{R}$ overall given by

$$E_b(n) = \lambda a. 2a_1 + 11(1 - \delta_{n-1}^{a_1}).$$

Special provision for the bilateral decision wait with a single column reducing to a rotated lateral decision wait as prescribed by [Equation 10.7](#) leads to

$$T_b(n) = \langle \lambda a. (\lambda i. ((A_b \ n) \ a)_i + (E_b \ n) \ a)^* t_2, \lambda a. (T_l \ 2) \langle 0, a_0 \rangle \rangle_{\delta_1^a}$$

as a total critical path specification. For $n = 1$, it devolves to a rotation of [Equation C.5](#), but otherwise is given by the sums of the absorption and emission phases.

General

A derivation of the critical paths in a general cascading decision wait proceeds similarly to the bilateral case subject only to a variable number of rows and to differences between the bicolunar and planar decision wait cells ([Figure 10.4](#) and [Figure 10.7](#)). An absorption phase critical path specification $A_p(s) : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ in terms of a function $A_p : \mathbb{N}^2 \rightarrow (\mathbb{N}^2 \rightarrow \mathbb{R}^2)$ parameterized by dimensions $s \in \mathbb{N}^2$ takes row and column indices $a = \langle a_0, a_1 \rangle \in \mathbb{N}^2$ to a list of absorption phase critical paths $(A_p \ s) \ a \in \mathbb{R}^2$. The row absorption phase critical path is given by $t_1 \in \mathbb{R}$ where

$$t = (T_l \ s_0) \langle 0, a_0 \rangle \in \mathbb{R}^2$$

is the total latency associated with column number a_0 on the rotated lateral decision wait visible in [Figure 10.6](#), whose number of columns matches the number of rows s_0 . To the column absorption phase critical path, the s_1 cells in any column but the last contribute two components each (a SHUNT and a TOGGLE) based on [Figure 10.7](#), or $2s_1(1 - \delta_{s_1-1}^{a_1})$. The MERGE network path $\tau(s_1, s_1 - 1 - a_1)$ is analogous to the bilateral case, and the remaining segment through the lateral decision wait is covered by $t_0 \in \mathbb{R}$ with t as above.

$$A_p(s) = \lambda a. (\lambda t. \langle t_1, 2s_1(1 - \delta_{s_1-1}^{a_1}) + \tau(s_1, s_1 - 1 - a_1) + t_0 \rangle) (T_l \ s_0) \langle 0, a_0 \rangle$$

To the emission phase critical path, the cells along row a_0 in each of the a_1 columns preceding the input column contribute two components each according to [Figure 10.7](#) as in the bilateral case. When the signal first reaches the cell in row number a_0 and column number a_1 during the emission phase (where a_1 is less than $s_1 - 1$), it traverses seven components in that cell on the path from ri to po , and then nine components from pi to po in each of the other $a_0 - 1$ cells that column, and then finally two more on the path from pi to d on row a_0 again, resulting in $9s_0(1 - \delta_{s_1-1}^{a_1})$ for a total emission phase $(E_p s) a \in \mathbb{R}$ given by

$$E_p(s) = \lambda a. 2a_1 + 9s_0(1 - \delta_{s_1-1}^{a_1})$$

suggesting an overall critical path specification

$$\lambda a. (\lambda i. ((A_p s) a)_i + (E_p s) a)^* \iota_2$$

except that the critical paths should reduce to those of a bilateral or lateral decision wait when the dimensions permit for consistency with [Equation 10.11](#).

$$T_p(s) = \begin{cases} \lambda a. (\lambda i. ((A_p s) a)_i + (E_p s) a)^* \iota_2 & \text{if } s_0 > 2 \wedge s_1 > 2 \\ \lambda a. (T_b s_0) \langle a_1, a_0 \rangle & \text{if } s_0 > 2 \wedge s_1 = 2 \\ \lambda a. (T_l s_0) \langle a_1, a_0 \rangle & \text{if } s_0 > 2 \wedge s_1 = 1 \\ \langle T_l s, T_b s \rangle_{\delta_2^{s_0}} & \text{otherwise} \end{cases} \quad (\text{C.6})$$

C.2.2 Quadrangular

The critical paths in a quadrangular decision wait depend on the particular decomposition $t \in \mathbb{S}$, so their specification $\dot{T}_q(t) : \mathbb{N}^* \rightarrow \mathbb{R}^*$ must be parameterized by it, with $\dot{T}_q : \mathbb{S} \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$ having to be defined as a fold

$$\dot{T}_q = \Lambda \lambda ((p, d), f). (\dot{\phi} p) \langle T_q(d, f), T_p \sum^* d \rangle_{\delta_e^f} \quad (\text{C.7})$$

in terms of some function

$$\dot{\phi} : \mathbb{N}^{**} \rightarrow ((\mathbb{N}^* \rightarrow \mathbb{R}^*) \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*))$$

yet to be determined that compensates for the effect of the permutations $p \in \mathbb{N}^{**}$ on the critical paths, some other function

$$T_q : \mathbb{N}^{*2} \times (\mathbb{N}^2 \rightarrow \mathbb{R}^2)^* \rightarrow (\mathbb{N}^2 \rightarrow \mathbb{R}^2)$$

also yet to be determined that takes the critical paths of the building blocks to the critical paths overall, and the function T_p defined by [Equation C.6](#) to handle terminal decompositions, which reduce to cascading forms by [Equation 10.30](#) assuming t satisfies $s = (\psi \Delta_q) t$ by [Equation 10.26](#) and [Equation 10.27](#) for some non-empty list of dimensions $s \in \mathbb{N}^2$.

Permutations

The function $\dot{\phi}$ is the easier part. Parameterized by a list of permutations $p \in \mathbb{N}^{**}$, it yields a function $\dot{\phi} p : (\mathbb{N}^* \rightarrow \mathbb{R}^*) \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$ that transforms a function $f : \mathbb{N}^* \rightarrow \mathbb{R}^*$ describing the critical paths in a decision wait prior to being permuted or rotated to a function

$$(\dot{\phi} p) f : \mathbb{N}^* \rightarrow \mathbb{R}^*$$

describing the critical paths in a decision wait having the desired dimensions s afterward. It may be helpful to think of ϕ as a function that performs the same operation as the function ϕ defined by Equation 10.25 in some sense, but operates on a more abstract representation of the circuit.

Part of this operation is to permute the inputs along the j -th axis of the decision wait according to the permutation p_{j+1} of the parameter p , and part of it is to rotate the decision wait by permuting the axes according to p_0 . Were it not for the latter rotation, a critical path starting from a_j -th input terminal along the j -th axis outside the input permutation network would pass through the terminal numbered $(p_{j+1})_{a_j}$ along that axis on the inside, so the function describing the critical paths of the result could be expressed as a composition

$$f \circ (\lambda a. (\lambda j. (p_{j+1})_{a_j})^* \iota_{|a|})$$

of the original function f with one that maps each term a_j of its argument a to its image with respect to p_{j+1} . On the other hand, if there were a rotation but no permutations, a critical path starting from the a_i -th terminal along the i -th axis on the outside would pass through the a_i -th terminal along the p_{0i} -th axis on the inside, so using the original f to get the right answer would require passing it an argument with a_i in the p_{0i} -th position and hence $a_{p'_i}$ in the i -th position, where p' is the inverse of p_0 (Section 8.1.5). A function capturing this effect could be expressed by a composition

$$f \circ ((\lambda p'. \lambda a. (\lambda i. a_{p'_i})^* \iota_{|p'|}) p_0^{-1})$$

of the original function f with one that does not alter any of the terms in its argument a but adjusts their order. Putting both of these effects together suggests the following definition.

$$\dot{\phi} = \lambda p. \lambda f. f \circ ((\lambda p'. \lambda a'. (\lambda i. a'_{p'_i})^* \iota_{|p'|}) p_0^{-1}) \circ \lambda a. (\lambda j. (p_{j+1})_{a_j})^* \iota_{|a|} \quad (\text{C.8})$$

Paths

Having compartmentalized the effects of any possible permutations and rotations as above, we can now focus on finding the critical paths through a quadrangular decision wait as if all inputs and outputs were ordered sequentially, but there is still a wealth of detail involved. As shown in Figure 10.9, any row input signal passes through a FORK, then through a row input on the routing stage x_1 , and then through a row input on one of the blocks in y , but the row input signal also passes concurrently through a MERGE network, a row input on x_1 , and a column input on the same inner block in y . Column input signals are analogous in the basic decomposition, but as explained in Section 10.3.2, x_0 disappears if the decomposition is vertical, and row input paths omit everything but x_1 if additionally the building block in y is lateral. A similar horizontal decomposition is a further possibility, as is a degenerate case noted in connection with Equation 10.18.

To calculate the critical paths starting from two inputs $a = \langle a_0, a_1 \rangle \in \mathbb{N}^2$, we can think of each input terminal a_i as being connected to the k_{0i} -th line of the k_{1i} -th bus along the i -th axis, with this bus having a width of k_{2i} lines. Then for example it would be possible to write $\tau(k_{2i}, k_{0i})$ by Equation C.4 for the segment of the path starting from the a_i -th terminal on the i -th axis due to the MERGE network. We can express

$$k_{0i} = (b (\lambda j. \iota_j)^* d_i)_{a_i}$$

in terms of the parameter $d \in \mathbb{N}^{*2}$ describing the decomposition, with

$$k_{1i} = ((\lambda j. j^{d_{ij}})^* \iota_{|d_i|})_{a_i}$$

$$k_{2i} = ((\lambda j. d_{ij}^{d_{ij}})^* \iota_{|d_i|})_{a_i}$$

and an additional parameter $k_{3i} = \delta_1^{|d_i|} \in \{0, 1\}$ indicating when the decomposition is vertical or horizontal. That is, k_{30} is non-zero when the decomposition is horizontal, and k_{31} is non-zero when the decomposition is vertical. Expressing all of them at once

$$\langle\langle k_{00}, k_{10}, k_{20}, k_{30} \rangle, \langle k_{01}, k_{11}, k_{21}, k_{31} \rangle\rangle = (\lambda i. (b (\lambda j. \iota_j)^* d_i)_{a_i} : (b (\lambda j. \langle j, d_{ij}, \delta_1^{|d_i|} \rangle^{\frac{d_{ij}}{d_i}})^* \iota_{|d_i|})_{a_i})^* \iota_2$$

would be more convenient as

$$k = e_0(d, a) \in (\mathbb{N}^2)^4$$

with $e_0 : \mathbb{N}^{*2} \times \mathbb{N}^2 \rightarrow (\mathbb{N}^2)^4$ defined by

$$e_0 = \lambda(d, a). ((\lambda i. (b (\lambda j. \iota_j)^* d_i)_{a_i} : (b (\lambda j. \langle j, d_{ij}, \delta_1^{|d_i|} \rangle^{\frac{d_{ij}}{d_i}})^* \iota_{|d_i|})_{a_i})^* \iota_2)^\top.$$

To work back from the end, each critical path concludes at one of the internal blocks y in [Figure 10.9](#) (assuming a decomposition that is neither vertical nor horizontal), whose specification is implicit in the parameter $f \in (\mathbb{N}^2 \rightarrow \mathbb{R}^2)^*$ in the context of [Equation C.7](#). Because f_0 refers to the routing stage x_0 and f_1 refers to x_1 in the figure, the item of f relevant to the terminals a would be one of the rest, or specifically $f_n : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ with

$$n = k_{10}|d_1| + k_{11} + 2$$

being simply its row major ordinal offset by two. Furthermore, the particular critical paths $b \in \mathbb{R}^2$ of interest would follow as

$$b = f_n k_0 \in \mathbb{R}^2$$

because $k_0 \in \mathbb{N}^2$ refers to the terminals in a numbered locally with respect to their buses as proposed above.

Just prior to the paths covered by b would be those segments passing through the routing stages labeled x_0 and x_1 in [Figure 10.9](#) (assuming a basic decomposition). The former receives a signal on the a_0 -th row and the k_{11} -st column because the column input a_1 belongs to the k_{11} -st bus. For analogous reasons, x_1 receives signals on row k_{10} and column a_1 , so the critical paths associated with them can be summarized as

$$r = \langle f_0 \langle a_0, k_{11} \rangle, f_1 \langle k_{10}, a_1 \rangle \rangle \in (\mathbb{R}^2)^2.$$

Because the paths b through the inner building block are in series with the paths r through the routing stages, the next step should be to add them. A signal starting from a_0 -th row input in a basic decomposition reaches a FORK first, and then propagates in parallel through two branches. One branch takes a path $r_{00} + b_0$ through the row input on the routing stage and the row input on the inner block, and the other branch takes a path $\tau(k_{20}, k_{00}) + r_{10} + b_1$ through a MERGE network, a row input on the other routing stage, and a column input on the inner block. Its overall critical path is therefore

$$1 + \max\{r_{00} + b_0, \tau(k_{20}, k_{00}) + r_{10} + b_1\}.$$

The critical path for a signal starting from the a_1 -st column input is analogous,

$$1 + \max\{r_{11} + b_1, \tau(k_{21}, k_{01}) + r_{01} + b_0\}$$

so the list of both of them would be

$$(\lambda i. 1 + \max\{r_{ii} + b_i, \tau(k_{2i}, k_{0i}) + (r_{1-i})_i + b_{1-i}\})^* \iota_2 \in \mathbb{R}^2$$

and the problem would be solved if the basic decomposition were the only possibility.

However, it is also necessary to consider the edge cases of vertical and horizontal decompositions. In a vertical decomposition, an input signal to the a_0 -th row could start with a FORK and then propagate in parallel through two paths as above, but because the routing stage x_0 would be absent, one of the paths would pass directly to the inner block and therefore reduce to b_0 while the other would remain $\tau(k_{20}, k_{00}) + r_{10} + b_1$ as above. The column input is not analogous in this case. Lacking a FORK or MERGE network, it would take only the path $r_{11} + b_1$ through the column inputs of the routing stage and the inner block, so the list of both paths would be

$$\langle 1 + \max\{b_0, \tau(k_{20}, k_{00}) + r_{10} + b_1\}, r_{11} + b_1 \rangle \in \mathbb{R}^2$$

if only it were that simple. There is also the possibility of the inner building block having only a single row as indicated by the condition $k_{20} = 1$, implying its absence from the construction. In this case the row and column paths start and end with the routing stage x_1 and thus reduce to $r_1 \in \mathbb{R}^2$. The correct result is therefore

$$\langle \langle 1 + \max\{b_0, \tau(k_{20}, k_{00}) + r_{10} + b_1\}, r_{11} + b_1 \rangle, r_1 \rangle_{\delta_1^{k_{20}}} \in \mathbb{R}^2$$

for a vertical decomposition, but at least the same reasoning yields the horizontal decomposition result for free.

$$\langle \langle r_{00} + b_0, 1 + \max\{b_1, \tau(k_{21}, k_{01}) + r_{01} + b_0\} \rangle, r_0 \rangle_{\delta_1^{k_{21}}} \in \mathbb{R}^2$$

The three different types of decompositions considered above are distinguishable by the parameter $k_3 \in \{0, 1\}^2$ as noted previously, with $k_{30} = 1$ for horizontal decompositions and $k_{31} = 1$ for vertical decompositions, so it is a short step to an expression $e_1(k, r, b) \in \mathbb{R}^2$ that covers all cases in terms of a function $e_1 : (\mathbb{N}^2)^4 \times (\mathbb{R}^2)^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$. Following Equation 10.18, we allow for a degenerate case devoid of routing stages as shown.

$$e_1 = \lambda(k, r, b). \begin{cases} b & \text{if } k_3 = \langle 1, 1 \rangle \\ \langle \langle 1 + \max\{b_0, \tau(k_{20}, k_{00}) + r_{10} + b_1\}, r_{11} + b_1 \rangle, r_1 \rangle_{\delta_1^{k_{20}}} & \text{if } k_3 = \langle 0, 1 \rangle \\ \langle \langle r_{00} + b_0, 1 + \max\{b_1, \tau(k_{21}, k_{01}) + r_{01} + b_0\} \rangle, r_0 \rangle_{\delta_1^{k_{21}}} & \text{if } k_3 = \langle 1, 0 \rangle \\ (\lambda i. 1 + \max\{r_{ii} + b_i, \tau(k_{2i}, k_{0i}) + (r_{1-i})_i + b_{1-i}\})^* \iota_2 & \text{if } k_3 = \langle 0, 0 \rangle \end{cases}$$

This expression enables a definition of T_q in the obvious way to meet the requirements of Equation C.7 and thereby account completely for critical paths in quadrangular decision waits.

$$T_q = \lambda(d, f). \lambda a. (\lambda k. e_1(k, \langle f_0 \langle a_0, k_{11} \rangle, f_1 \langle k_{10}, a_1 \rangle \rangle), (\lambda n. f_n k_0) k_{10} |d_1| + k_{11} + 2)) e_0(d, a)$$

C.2.3 Dendriform

Moving on to dendriform decision waits, we seek a description of the critical paths in terms of a function $\dot{T}_d : \mathbb{S} \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$ defined similarly to Equation C.7 as a fold

$$\dot{T}_d = \Lambda \lambda((p, d), f). (\dot{\phi} p) \langle T_d(d, f), T_p \sum^* d \rangle_{\delta_\epsilon^f} \quad (\text{C.9})$$

operating on a decomposition $t \in \mathbb{S}$ satisfying $s = \psi \Delta_d t$ for some non-empty list of positive dimensions $s \in \mathbb{N}^*$ by Equation 10.26 and Equation 10.31 (the defining property of a dendriform

decomposition). The functions $\dot{\phi}$ defined by Equation C.8 and T_p defined by Equation C.6 serve the same purpose in Equation C.9 as in Equation C.7, and a function

$$T_d : \mathbb{N}^{**} \times (\mathbb{N}^* \rightarrow \mathbb{R}^*)^* \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$$

transforming the critical path specifications of the building blocks to that of the result is yet to be determined. T_d could be useful by itself for calculating the critical paths in multidimensional decision waits restricted either to dendriform and cascading decompositions, or otherwise as an intermediate step toward the analysis of more general forms considered in Section C.2.5.

Critical paths in a dendriform decision wait are a more straightforward proposition than those of quadrangular decision waits because there are no exposed concurrent paths and no edge cases. As shown in Figure 10.16, there is a front end consisting of any number multidimensional decision wait building blocks each connected directly to a back end consisting of just one. Each path starts through one of the front end blocks and finishes through the only back end block. In the context of Equation C.9, f_i describes the i -th front end block for $0 \leq i < |d|$, and $f_{|d|}$ describes the back end block, where $t = ((p, d), v) \in \mathbb{S}$ is the decomposition, all of which are known already by hypothesis if f is non-empty. To find a whole critical path starting from any particular terminal, we have to calculate the segment of the path through the relevant front end block using f_i for the appropriate i , and then add the rest of the path as given by $f_{|d|}$ to that result.

The problem now reduces to finding the right arguments to plug into f_i and $f_{|d|}$ to obtain the list of critical path lengths

$$T_d(d, f) a \in \mathbb{R}^{|b d|}$$

associated with a list of input terminal addresses

$$a \in \mathbb{N}^{|b d|}$$

whose length $|b d|$ presumably matches the number of dimensions perceived externally, whereas each function $f_i : \mathbb{N}^{|d_i|} \rightarrow \mathbb{R}^{|d_i|}$ pertains to a decision wait having only $|d_i|$ of the total $|b d|$ dimensions. Each dimension corresponds to an external input bus such that the first front end block is connected to the first $|d_0|$ buses, the next front end block to the next $|d_1|$ buses, and so on. The appropriate argument $c_i \in \mathbb{N}^{|d_i|}$ to f_i determining its effect on $T_d(d, f) a$ is therefore given by

$$c = (\lambda i. (a \ll |b(d \mid i)|) \mid |d_i|)^* \iota_{|d_i|} \in \mathbb{N}^{*|d|}$$

which is just an unflattened form of a satisfying $a = b c$ and $|c_i| = |d_i|$.

Finding the right argument to plug into $f_{|d|}$ is a bit more complicated. The back end decision wait has $|d|$ dimensions with $\prod d_n$ inputs in the n -th dimension for $0 \leq n < |d|$. The inputs to the n -th dimension come by a bus of width $\prod d_n$ from the outputs of the n -th front end block. If the n -th front end block receives input signals on terminals numbered c_n , it emits a signal on an output terminal numbered

$$(\bar{t}_{d_n})^{-1} c_n \in \mathbb{N}$$

where $\bar{t}_{d_n} \in (\mathbb{N}^{|d_n|})^{\prod d_n}$ is a list of lists of coordinates by Equation 10.3, $(\bar{t}_{d_n})_v \in \mathbb{N}^{|d_n|}$ is the list of coordinates of the lexicographically v -th point in a $|d_n|$ -dimensional lattice, and for each $0 \leq m < |d_n|$, the value of the coordinate $(\bar{t}_{d_n})_{vm}$ ranges from 0 to $d_{nm} - 1$. Evaluating the list of critical paths through the back end obtained when the inputs to the front end are determined by c

$$f_{|d|} (\lambda n. (\bar{t}_{d_n})^{-1} c_n)^* \iota_{|d|} \in \mathbb{R}^{|d|}$$

yields among others the particular critical path $(f_{|d|} (\lambda n. (\bar{t}_{d_n})^{-1} c_n)^* \iota_{|d|})_i$ originating from some terminal along the i -th axis of the back end block. This path necessarily forms the latter segment of any path passing through the i -th front end block, because all outputs from the i -th front end block ride the same bus.

Carrying this insight to its conclusion, we can have a list of critical paths associated with the i -th front end block

$$f_i c_i \in \mathbb{R}^{|d_i|}$$

with each path to be extended by a common back end segment

$$(\lambda j. (f_i c_i)_j + (f_{|d|} (\lambda n. (\bar{t}_{d_n})^{-1} c_n)^* \iota_{|d|})_i)^* \iota_{|d_i|}$$

for any front end block numbered $0 \leq i < |d|$

$$(\lambda i. (\lambda j. (f_i c_i)_j + (f_{|d|} (\lambda n. (\bar{t}_{d_n})^{-1} c_n)^* \iota_{|d|})_i)^* \iota_{d_i})^* \iota_{|d|}$$

combined into a list $e_2(d, f) c \in \mathbb{R}^{|b d|}$ with the function

$$e_2 : (\mathbb{N}^{**} \times (\mathbb{N}^* \rightarrow \mathbb{R}^*)^*) \rightarrow (\mathbb{N}^{**} \rightarrow \mathbb{R}^*)$$

defined as

$$e_2 = \lambda(d, f). \lambda c. b (\lambda i. (\lambda j. (f_i c_i)_j + (f_{|d|} (\lambda n. (\bar{t}_{d_n})^{-1} c_n)^* \iota_{|d|})_i)^* \iota_{d_i})^* \iota_{|d|}$$

suggesting the following definition for T_d to complete [Equation C.9](#).

$$T_d = \lambda(d, f). e_2(d, f) \circ \lambda a. (\lambda i. (a \ll |b(d \mid i)|) \mid |d_i|)^* \iota_{|d|}$$

C.2.4 Crossbar

Analyzing the critical paths for a crossbar decision wait is only moderately more complicated than for a dendriform decision wait. A function $\dot{T}_c : \mathbb{S} \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$ given by

$$\dot{T}_c = \Lambda \lambda((p, d), f). (\dot{\phi} p) \langle T_c(d, f), T_p \sum^* d \rangle_{\delta_c} \quad (\text{C.10})$$

is defined similarly to the function \dot{T}_d ([Equation C.9](#)) in terms of $\dot{\phi}$, T_p , and a function

$$T_c : \mathbb{N}^{**} \times (\mathbb{N}^* \rightarrow \mathbb{R}^*)^* \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$$

to be derived presently, but operates on decompositions $t \in \mathbb{S}$ satisfying $s = \psi \Delta_c t$ for some non-empty $s \in \mathbb{N}^*$ by [Equation 10.33](#).

As a reminder, [Figure 10.17](#) shows that a crossbar decision wait with a decomposition $t \in \mathbb{S}$ having a root $(p, d) \in \mathbb{N}^{**} \times \mathbb{N}^{*2}$ has $|d_1|$ front end building blocks and $\prod d_0$ back end building blocks, as well as a FORK network in front of the front end. Each front end block has $|d_0| + 1$ dimensions, each back end block has $|d_1|$ dimensions, and the combination is equivalent to a decision wait with $|d_0| + |d_1|$ dimensions, but only the signals in the first $|d_0|$ dimensions pass through the FORK network. In the context of [Equation C.10](#), the first $|d_1|$ terms of f are the functions describing the critical paths in the front end, and the remaining $\prod d_0$ terms describe the critical paths in the back end blocks.

A selection of input terminals numbered $a \in \mathbb{N}^{|d_1|}$ with each a_j being the number of the terminal in the j -th dimension to receive an input signal would mean each front end block receives all of the first $|d_0|$ signals specified by a via the FORK network, and exactly one of the latter $|d_1|$. This condition implies that a path of length $(u_i)_j \in \mathbb{R}$ through the i -th front end block given by

$$u = \lambda i. f_i((a \uparrow |d_0|) \parallel \langle a_{|d_0|+i} \rangle) * \iota_{|d_1|}$$

must be traversed concurrently through every front end block by the j -th signal for any j among the first $|d_1|$ specified by a , but only the single path $(u_j)_{|d_0|}$ through the j -th front end block is traversed by signals in dimensions $j \geq |d_1|$.

On the other hand, only the n -th of the $\prod |d_0|$ back end blocks receives any signals at all, with n given by

$$n = (\bar{\iota}_{d_0})^{-1}(a \uparrow |d_0|)$$

according to Equation 10.3. Each front end block has the same number $\prod |d_0|$ of output buses, each of the $\prod |d_0|$ back end blocks is connected to exactly one bus from each of the front end blocks, and the front end inputs in the first $|d_0|$ dimensions suffice to narrow the output down to the same output bus on all of the front end blocks. The position of this bus relative to the others from the same front end block would be the lexicographic ordinal of $a \uparrow |d_0|$ relative to the range of $\bar{\iota}_{d_0}$, the set of possibilities within the given dimensions.

Having identified the back end block relevant to the input list a as the n -th, we have also fully determined the relevant list of critical paths

$$v = f_{|d_1|+n}(a \ll |d_0|) \in \mathbb{R}^{|d_1|}$$

through the back end block in terms of the function $f_{|d_1|+n}$ specifying them in the context of Equation C.10.

With the front end and the back end covered, there is only the FORK network left. Every input in each of the first $|d_0|$ input buses passes through a FORK with one output for each of the $|d_1|$ front end blocks. Equation C.4 is just as applicable to FORK networks as to MERGE networks, so it is reasonable to write $\tau(|d_1|, a_j)$ as a component of the path followed by the signal to the a_j -th terminal in the j -th dimension for values of j less than $|d_0|$.

Obtaining the list of combined critical paths $T_c(d, f) a \in \mathbb{R}^{|a|}$ is now a matter of adding the segments together appropriately. After going through the path $\tau(|d_1|, a_j)$ due to the FORK network, a signal originating from the a_j -th terminal in the j -th dimension with $j < |d_0|$ takes the j -th path through the i -th front end block followed by the i -th path through the back end block

$$(u_i)_j + v_i$$

concurrently for all $|d_1|$ front end blocks, implying a critical path

$$\max \mathcal{R}((\lambda i. (u_i)_j + v_i) * \iota_{|d_1|})$$

along the segment through the front and back ends, or a critical path of

$$\tau(|d_1|, a_j) + \max \mathcal{R}((\lambda i. (u_i)_j + v_i) * \iota_{|d_1|})$$

overall. Signals originating on the a_j -th terminal in the j -th dimension for $j \geq |d_0|$ bypass the FORK network and take the last path through the j -th front end block followed by the j -th path through the back end block.

$$(u_j)_{|d_0|} + v_j$$

An expression for the list critical paths taken by all $|d_0| + |d_1|$ signals would be

$$(\lambda j. \tau(|d_1|, a_j) + \max \mathcal{R}((\lambda i. (u_i)_j + v_i)^* \iota_{|d_1|}))^* \iota_{|d_0|}) \parallel (\lambda j. (u_j)_{|d_0|} + v_j)^* \iota_{|d_1|}$$

which is easier to abbreviate as $e_3(u, v, d, a)$ in terms of a function $e_3 : \mathbb{R}^{**} \times \mathbb{R}^* \times \mathbb{N}^{*2} \times \mathbb{N}^* \rightarrow \mathbb{R}^*$ defined by

$$e_3 = \lambda(u, v, d, a). ((\lambda j. \tau(|d_1|, a_j) + \max \mathcal{R}((\lambda i. (u_i)_j + v_i)^* \iota_{|d_1|}))^* \iota_{|d_0|}) \parallel (\lambda j. (u_j)_{|d_0|} + v_j)^* \iota_{|d_1|})$$

so that the desired result $T_c(d, f) : \mathbb{N}^* \rightarrow \mathbb{R}^*$ to complete [Equation C.10](#) follows from a function

$$T_c : \mathbb{N}^{*2} \times (\mathbb{N}^* \rightarrow \mathbb{R}^*)^* \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$$

defined as

$$T_c = \lambda(d, f). \lambda a. e_3(\lambda i. f_i((a \mid |d_0|) \parallel \langle a_{|d_0|+i} \rangle)^* \iota_{|d_1|}, (\lambda n. f_{|d_1|+n}(a \ll |d_0|)) (\bar{\iota}_{d_0})^{-1} (a \mid |d_0|), d, a).$$

C.2.5 General

A critical path analysis of any combination of the decompositions investigated in previous sections is possible without much further effort. Any decomposition $t \in \mathbb{S}$ satisfying $s = (\psi \Delta_g) t \neq \epsilon$ by [Equation 10.35](#) determines a decision wait by [Equation 10.36](#) with $|s|$ input buses. For any list of terminal numbers $a \in \mathcal{R}(\bar{\iota}_s)$, transmitting $|a|$ signals concurrently such that the i -th input bus receives a signal on its a_i -th terminal for all $0 \leq i < |a|$ implies that the signal on the i -th bus takes a critical path of length $((T_g t) a)_i \in \mathbb{R}$ to the terminal that emits an output signal as given by a function

$$T_g : \mathbb{S} \rightarrow (\mathbb{N}^* \rightarrow \mathbb{R}^*)$$

defined by the recurrence

$$T_g = \Lambda \lambda((p, d), f). (\dot{\phi} p) \begin{cases} \langle 0, T_p \sum^* d \rangle_{\delta_2^{|d|}} & \text{if } |f| = 0 \\ T_d(d, f) & \text{if } |f| = |d| + 1 \\ T_c(d, f) & \text{if } |f| = |d_1| + \prod d_0 \\ T_q(d, f) & \text{if } |f| = (\sum d_0)(\sum d_1) + 2 \end{cases} \quad (\text{C.11})$$

in terms of functions $\dot{\phi}$, T_p , T_d , T_c , and T_q defined previously. Each case in this definition pertains to a particular type of decomposition distinguishable from the others by its dimensions and number of subtrees.

As noted at the beginning of [Section C.2](#), a function like $T_g t : \mathbb{N}^* \rightarrow \mathbb{R}^*$ representing every possible combination of critical paths pertaining to just one individual decomposition t might be too detailed for the final analysis. Although a description in this form has been necessary as a stepping stone, there are countless ways to reduce it subsequently to a point estimate. For example, a metric $\tilde{T} : \mathbb{S} \rightarrow \mathbb{R}$ useful for comparing the worst case critical paths between two decompositions is easy to define in terms of T_g as follows

$$\tilde{T} = \lambda t. \max \mathcal{R}(b (T_g t)^* \bar{\iota}_{(\psi \Delta_g) t})$$

assuming no path is any more important than another. A simple variation $\bar{T} : \mathbb{S} \rightarrow \mathbb{R}$ facilitates comparison of average critical paths assuming all combinations of inputs are equally probable.

$$\bar{T} = \lambda t. \left(\lambda m. \frac{\sum m}{|m|} \right) b (T_g t)^* \bar{\iota}_{(\psi \Delta_g) t} \quad (\text{C.12})$$

Multidimensional anxiety theory

1. Trace the paths taken by all signals through the bilateral and planar decision wait cells (Figure 10.4 and Figure 10.7) during the absorption and emission phases using highlighter pens or similar technology. Where do the numbers nine, seven, and two mentioned on page 594 come from?
2. What small changes to the definitions of the critical path metrics would make them more like a measure of power consumption?
3. Define a metric $T : \mathbb{S} \rightarrow \mathbb{R}$ in terms of T_g to minimize the standard deviation of the critical path. Would it ever be useful for anything?
4. Define a metric in terms of M_g and T_g suitable for optimizing the average critical path subject to a fixed maximum cost $K \in \mathbb{R}$. (hint: Treat infinity as a number.)
5. Generalize \bar{T} from Equation C.12 to non-uniform distributions and adjust the previous two solutions similarly.
6. Confirm or refute the claim in the first paragraph on page 587 that crossbar decision waits always have a lower component count than dendriform decision waits of similar dimensions.
7. Solve item 5, page 313 and incorporate the results into the component count and critical paths metrics M_g and T_g .
8. Extend these results to sparse decision waits as defined in Chapter 11.
9. Rework everything with arbitrary fixed costs and critical path lengths depending on the type of primitive component as in Figure C.1.



Glance into the world just as though
time were gone, and everything
crooked will become straight to you.

Friedrich Nietzsche



LATENCY ARITHMETIC

There is an important distinction to be made between critical paths as considered in [Appendix C](#) and the more practical metric of latency. A critical path is defined for our purposes as the number of components, or at best a weighted sum of the number of components, traversed by a signal from an input to an output terminal, whereas the latency is the elapsed time between a signal being received on the input terminal and a signal appearing on the output. Critical paths are easy to work out by doing arithmetic. Latencies are difficult because they depend on wire propagation delays, which depend on environmental conditions, technological process variations, placement, and routing. Latencies are the real performance data, and critical path lengths are just a substitute.

In addition to those difficulties, there are philosophical or at least methodological problems with the concept of latency. For example, signals transmitted simultaneously to multiple terminals on a decision wait result in only a single output signal, so should all of the terminals be judged to have identical latencies? If not, should we envision a laboratory test of latency whereby every terminal other than the one being tested has received its signal in the distant past? Would such a test be relevant to actual operating conditions or be of any help in deriving more general results?

Because it is unsatisfying to retreat from these questions completely, this appendix delves briefly¹ into a speculative concept of critical path length calculations based on an attempt to model the technological substrate and routing algorithm characteristics statistically in aggregate.

D.1 Latencies as a vector space

Latencies can be modeled by a pair of non-negative real numbers $(t, w) \in \mathbb{R} \times \mathbb{R}$. The right side w models the portion of the latency attributable to the distance traversed by the signals along the internal or external wires *en route* between the terminals, and the left side models the portion of the latency due to whatever else takes any time in addition to that. In principle, these parameters could be empirically determined for a pair of terminals as follows.

¹especially if it turns out to be misguided

1. Apply a test signal to the input terminal.
2. Measure the elapsed time d taken for a signal to appear on the output terminal.
3. Measure the spatial separation x between the relevant terminals.
4. Measure the time w needed by a signal to propagate through a wire of length x .
5. Designate a latency of $(d - w, w)$ for the terminal pair.

Latencies form a vector space with scalar multiplication defined in the obvious way for a scalar $k \in \mathbb{R}$,

$$k(t, w) = (t, w)k = (kt, kw)$$

and with the **magnitude** of a latency (t, w) denoted $|(t, w)|$ and defined as $t + w$. Addition of latencies is given by

$$(t_a, w_a) \oplus (t_b, w_b) = (t_a + t_b, w_a + w_b).$$

and represents the combined latency of two circuits with latencies (t_a, w_a) and (t_b, w_b) respectively connected in series.

D.2 Comparison of latency vectors

This style of maintaining latencies as two-dimensional vectors could be relevant to situations when two signals propagate separately from a FORK and converge subsequently on a JOIN, thereby obliging us to estimate the effective latency of the combination. If wire delays were neglected, the maximum of the two latencies would be an adequate metric as noted in [Appendix C](#), but in general other considerations may apply. Based on some physical intuition and a few modest assumptions sketched below, the effective latency of two synchronized parallel paths seems likely to exceed what their magnitudes alone would suggest.

D.2.1 Manhattan distances

For circuits laid out on a planar surface, routing algorithms tend to use a rectangular coordinate system and restrict the connecting wires to two orthogonal directions. A connection between points (i, j) and (h, k) can follow the L-shaped path $(i, j) - (i, k) - (h, k)$, or the opposite L-shaped path via (h, j) , or an intermediate path with multiple L-shaped turns, for a total length of $|i - h| + |j - k|$ in all cases, but can never take a direct diagonal line. This quantity represents the so called **Manhattan distance** between the two points, and is a more appropriate metric than the ordinary Euclidean distance (*i.e.* the distance “as the crow flies”) because it indicates the actual distance a signal has to travel to get from one point to the other by way of an optimally routed wire.

Under the Manhattan distance metric, the locus of points on a plane equidistant from an origin forms a rhombus as shown in [Figure D.1](#) instead of a circle. If a path has a latency of (t, w) and it is laid out without any extra wires introduced between the components, the end point could be as far as w/c directly above or below the starting point, or w/c to the left or to the right of it, or anywhere within the convex hull of these extremities, where c is the rate of signal propagation through a wire, but we assume a normalized value of $c = 1$ hereafter. In practice, there may be extra wire lengths needed between the components along a path if they can not all be placed in direct physical proximity, so we should regard the latency estimate resulting from this analysis as a lower bound.

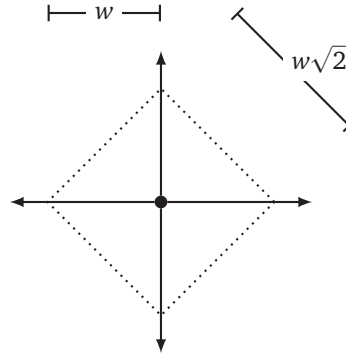


Figure D.1: The locus of points at a Manhattan distance w from the origin forms a rhombus with sides of length $w\sqrt{2}$.

D.2.2 Expected separations

Lacking any more specific information about the routing algorithm, suppose the end point of a path starting at the origin can lie anywhere within this rhombic region with equal probability (*i.e.*, according to a uniform distribution), and because the rate of propagation c is normalized to unity, let wire propagation delays serve as proxies for distances. Two paths with latencies (t_a, w_a) and (t_b, w_b) respectively originating at the same FORK therefore terminate within concentric rhombi with edge lengths of $w_a\sqrt{2}$ and $w_b\sqrt{2}$ respectively.

All placement and routing algorithms are constrained to refrain from putting two things in the same place, so the paths will run in different directions generally and the end point locations will be anti-correlated (*i.e.*, more likely to be far from each other than near). A lower bound on their expected separation is obtained therefore by assuming uncorrelated uniformly distributed endpoints at Manhattan distances of no more than w_a and w_b from the origin. We can simplify the calculations by rotating the coordinate system so that the x and y coordinates of each end point are uniformly distributed between plus and minus $w_a/\sqrt{2}$ or $w_b/\sqrt{2}$ as the case may be. This construction implies an expected Manhattan distance of $\bar{S}(w_a, w_b)$ between the end points as follows.²

$$\bar{S}(w_a, w_b) = \left(\lambda(a, b) \cdot \frac{a^2 + 3b^2}{3\sqrt{2}b} \right) (\min\{w_a, w_b\}, \max\{w_a, w_b\}) \quad (\text{D.1})$$

D.2.3 Expected wire delays

This separation distance must still be negotiated by the signals after they reach the ends of their respective paths before they can synchronize at a JOIN. The routing algorithm can do no better on average than to connect the end points by a wire of length $\bar{S}(w_a, w_b)$, so if the signals were to emerge simultaneously from the end points, they would require an additional time of at least $\bar{S}(w_a, w_b)/2$ to converge. More generally, one signal reaches its end point $|(t_a, w_a)| - |(t_b, w_b)|$ earlier than the other (*i.e.*, a time interval equal to the absolute difference in latency magnitudes) and has that much of a head start to their common destination. If this interval exceeds $\bar{S}(w_a, w_b)$,

²See [193] for an algorithmic approach to manipulating probability distributions.

then it is as if the signal on the shorter path traverses the whole separation between the end points before the signal on the longer path emerges, and the entire wire propagation delay reduces to the greater of w_a or w_b . Accordingly, we can estimate the wire propagation delay as

$$\bar{D}((t_a, w_a), (t_b, w_b)) = \max\{w_a, w_b\} + \max\{\bar{S}(w_a, w_b) - \left| |(t_a, w_a)| - |(t_b, w_b)| \right|, 0\}/2.$$

D.3 Parallel combination of latency vectors

Based on the results above, it is convenient to use this notation to represent the combined latency of two concurrent paths with latencies of (t_a, w_a) and (t_b, w_b) .

$$(t_a, w_a) \diamond (t_b, w_b) = (\max\{t_a, t_b\}, \bar{D}((t_a, w_a), (t_b, w_b))) \quad (\text{D.2})$$

An expression of the form $x \diamond y$ can be read informally as the effective latency of two synchronized parallel paths with latencies x and y plus a conservative estimate of the additional latency due to the wire delays. Hence, it satisfies

$$|x \diamond y| \geq \max\{|x|, |y|\}$$

and strict equality holds whenever the wire delay components of x and y are both zero.

Latent labors

1. Verify [Equation D.1](#) using whichever of the following techniques is least suspicious.

- a) a formal proof
- b) a Monte Carlo simulation
- c) a computer algebra system



2. Do these identities hold? Should they?

- a) $x \diamond y = y \diamond x$
- b) $(x \diamond y) \diamond z = x \diamond (y \diamond z)$

3. Can the Manhattan distance between two points change when the coordinate system is rotated, and if so, how is it justifiable to rotate the coordinate system when deriving [Equation D.1](#)?

4. All the cool kids are using secret proprietary routing algorithms that allow wires to be routed diagonally. Derive the equivalent to [Equation D.1](#) assuming a Euclidean distance metric by breaking it down like this:

- a) A constant valued probability density function corresponding to a uniform distribution over a disk of radius R centered at the origin and parameterized by polar coordinates has the form

$$\lambda(r, \theta) = \begin{cases} 1/(\pi R^2) & \text{if } 0 \leq r \leq R \\ 0 & \text{otherwise.} \end{cases}$$

Express the probability of a random draw being a distance r from the origin such that $a \leq r \leq b$ holds for constants $0 \leq a \leq b \leq R$. Are all annuli of width $b - a$ equally probable?

- b) Infer an expression for a univariate probability density function f_R whose value at r is proportional to the probability mass of an annulus of infinitesimal width dr and radius r centered at the origin when the disk has radius R . In other words, find a function f_R to make the answer to part a) coincide with

$$\int_{r=a}^b f_R(r) dr.$$

- c) Given the expected angular separation of $\pi/2$ and the Pythagorean theorem, find the expected Euclidean distance between two rectangular coordinates $(a, 0)$ and $(0, b)$, where a is distributed according to f_{w_a} and b is distributed according to f_{w_b} derived above for arbitrary $w_a, w_b > 0$, as shown in [Figure D.2](#). (hint: If the expected distance does not have an analytical solution, maybe the expected square of the distance does. Try for upper and lower bounds.)

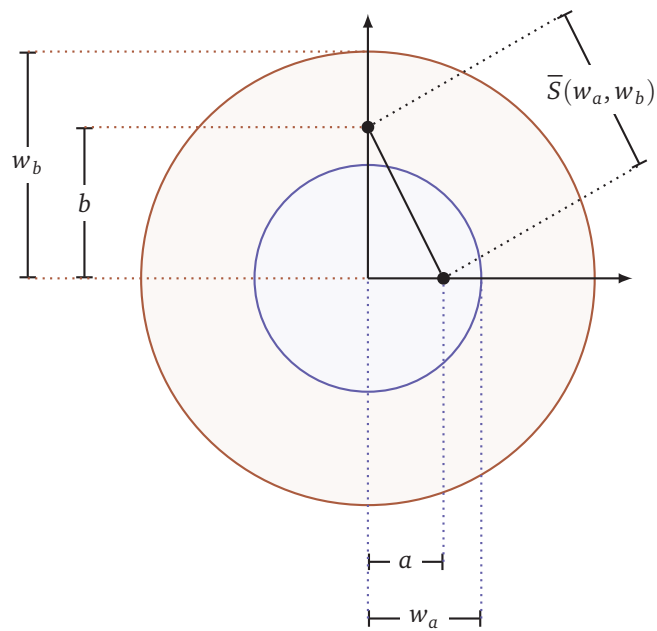


Figure D.2: Two sample points $(a, 0)$ and $(0, b)$ are distributed along their respective axes according to probability density functions f_{w_a} and f_{w_b} . What is the expected Euclidean (not Manhattan) distance separating them?

It is far better to foresee even
without certainty than not to foresee
at all.

Henri Poincaré



ARBITER METRICS

This appendix is meant as a primer for readers who are serious about optimizing their arbiters. As explained in [Chapter 12](#), any measure $(Q, t) \in \mathbb{R} \times \mathbb{R}$ of an arbiter with a decomposition $t \in \mathbb{A}$ and an arity n can be averaged over all request vectors $r \in \{0, 1\}^n$ with respect to anticipated access statistics ν as $(\bar{\Theta}_n \nu)(Q, t) \in \mathbb{R}$ by plugging ν , Q , and t into [Equation 12.47](#), but finding the right function Q to capture an intended property is an art. The interesting examples normally take the form of a recurrence over t . Two variations on this theme, possibly useful in themselves or in combination with others of the reader's invention, concern contention and critical path length. Starting with the easier one, these metrics are constructed respectively in [Section E.1](#) and [Section E.2](#). The basic techniques exemplified here might then serve as templates for further customization.

E.1 Contention

Contention is what happens when multiple requests arrive concurrently at the same arbiter, which as always can grant only one. Contended requests take longer to grant than uncontended requests due to the time needed to resolve them, with the latency attaining a (probabilistic) maximum when requests are perfectly synchronized and decaying asymptotically to the uncontended latency as the time between requests increases.¹ However, a physically realistic model of this phenomenon would be overkill for purposes of comparison if a more easily computed metric were found to imply the same design choices.

A way of quantifying contention starts by envisioning each request as being subject individually to some variable amount of it. We should expect a request propagating through an arbiter to encounter no contention if there are no other requests concurrent with it, moderate contention in light traffic, and significant contention in heavy traffic. A proxy for contention that fits with these expectations is the probability of a competing request to the same arbiter as the request in question, where a competing request with high probability raises the score more than one with low probability.

¹or not, depending on who is right [[191](#), [195](#)]

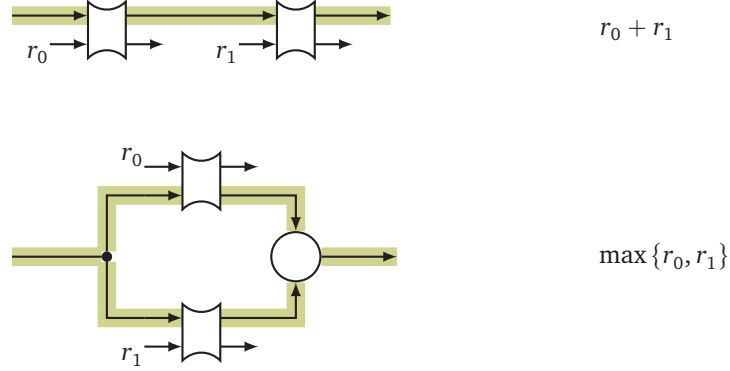


Figure E.1: Contention along the highlighted path is estimated as the sum of the competing request probabilities encountered in series (above) or their maximum in parallel (below).

That is, if an arbiter has two ports, then the measure of contention assigned to the path through either one of the ports is the probability of a request on the *other* port.

What should be the effect of multiple competitions with other requests in multiple stages? A measure of contention commensurate with its effect on latency should follow the same rules illustrated in Figure C.1 for critical paths, specifically pertaining to multiple sites of contention as shown in Figure E.1. That is, parallel contended requests are only as time consuming as the more severe of the two, whereas contended requests in series are resolved sequentially and therefore combine additively.

Both of these rules are needed to evaluate the list of contention estimates manifested externally by the more complex arrangement of arbiters in a mesh described by parameters

$$(b, m) \in \mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}$$

as proposed on page 370. Given $c \in \mathbb{R}^{*|b|m|}$ with $c_{kl} \in \mathbb{R}$ measuring the contention on the k -th port of the arbiter corresponding to $(b, m)_l$, we could split c into

$$c' = (\lambda i. c \circ \iota_{|m_i|}^{b(m_i)})^* \iota_{|m|} \in \mathbb{R}^{***} \quad (\text{E.1})$$

by stages so that $((h_1 c') (m, s))_i$ repurposes the function h_1 defined by Equation 12.30 to express the contention faced in the i -th stage by the request externally numbered s . Collecting the list

$$\iota_{b_{sz}}^{\sum(b_s, z)} \in \mathcal{R}(\iota_{|m|})^*$$

of stage numbers i within the zone numbered z of the signal numbered s allows for an expression

$$((h_1 c') (m, s))^* \iota_{b_{sz}}^{\sum(b_s, z)} \in \mathbb{R}^{|b_{sz}|}$$

listing the contention estimates through all parallel paths across z for s , and hence the estimate

$$\max \mathcal{R}(((h_1 c') (m, s))^* \iota_{b_{sz}}^{\sum(b_s, z)}) \in \mathbb{R}$$

of contention contributing to the total

$$\sum_{z=0}^{|b_s|-1} \max \mathcal{R}(((h_1 c') (m, s))^* \iota_{b_{sz}}^{\sum(b_s \iota z)})$$

encountered by the signal s sequentially over all zones. Then the list $X_0(c, (b, m)) \in \mathbb{R}^{|b|}$ of externally visible contentions indexed by signal numbers s is expressible in terms of a function

$$X_0 : \mathbb{R}^{**} \times (\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}) \rightarrow \mathbb{R}^*$$

defined as follows.

$$X_0 = \lambda(c, (b, m)). \left(\lambda s. \sum_{z=0}^{|b_s|-1} \max \mathcal{R}(((h_1 (\lambda i. c \circ \iota_{|m_i|}^{b(m \iota i)})^* \iota_{|m_i|}) (m, s))^* \iota_{b_{sz}}^{\sum(b_s \iota z)}) \right)^* \iota_{|b|}$$

Compared to the mesh arbiter, evaluating the contention of other forms is easy. A dendriform arbiter with arity n and decomposition $d \in \nabla_d n$ according to [Equation 12.19](#) has the combined contention metric

$$\sum^* \langle b(c \iota |d_0|), c_{|d_0|} \rangle^T \in \mathbb{R}^n$$

implied by the straightforward series combination of the leaves with the root and $c \in (\mathbb{R}^*)^{|d_0|+1}$ as their contention metrics. The token ring is also easy, with its combined contention

$$b(\lambda i. c_i \ll 1)^* \iota_{|c|} \in \mathbb{R}^n$$

obtained from that of its constituent arbiters $c \in (\mathbb{R}^*)^{|d_0|}$ for a decomposition $d \in \nabla_o n$ according to [Equation 12.23](#) by taking all but the first term of each term of c . (The first terms correspond to the internal 2Φ handshaking ports between cells shown in [Figure 12.9](#) that are not visible externally.) For a wire or primitive arbiter there would be no relevant value of c , but the contention metric is known directly from the request probability vector $r \in [0, 1]^*$ as $\langle 0 \rangle$ or $\langle r_1, r_0 \rangle$ respectively (that is, with no contention through a wire, and the opposite request probabilities through a primitive arbiter). An expression $X_1(c, d, r) \in \mathbb{R}^*$ for the resulting contention from any compatible values of c , d , and r could be given in terms of a function

$$X_1 : \mathbb{R}^* \times (\mathbb{S}^* \cup (\mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**})) \times [0, 1]^* \rightarrow \mathbb{R}^*$$

defined as follows.

$$X_1 = \lambda(c, d, r). \begin{cases} \langle \sum^* \langle b(c \iota |d_0|), c_{|d_0|} \rangle^T, b(\lambda i. c_i \ll 1)^* \iota_{|c|} \rangle_{|d|-1} & \text{if } d \in \mathbb{S}^{**} \\ \langle X_0(c, d), \langle \langle 0 \rangle, \langle r_1, r_0 \rangle \rangle_{|r|-1} \rangle_{\delta_c} & \text{otherwise} \end{cases}$$

The contention metric for a decomposition $t = ((p, d, k), \nu) \in \hat{\mathbb{A}}$ and a corresponding request probability vector $r \in [0, 1]^{|p|}$ is obtained as $X_1(c, d, r \circ p)$ with $c \in \mathbb{R}^{*|\nu|}$ being the result recursively obtained for the subtrees ν with request probability vectors

$$\hat{R}(d, k, \hat{H}_1^* \nu) (r \circ p) \in [0, 1]^{*|\nu|}$$

in terms of the incremental transfer function \dot{H}_1 defined on page 397 and the request propagation function \dot{R} explained on page 399. To express the recurrence precisely, let $X_2 : \dot{\mathbb{A}} \rightarrow ([0, 1]^* \rightarrow \mathbb{R}^*)$ defined as

$$X_2(t) = \lambda r. (\lambda((p, d, k), v). X_1((X_2^* v) \triangle (\dot{R}(d, k, \dot{H}_1^* v) (r \circ p)), d, r \circ p)) t$$

denote the function that takes a decomposition $t \in \dot{\mathbb{A}}$ to a function $X_2 t : [0, 1]^* \rightarrow \mathbb{R}^*$ taking a request probability vector $r \in [0, 1]^*$ to a list $(X_2 t) r \in \mathbb{R}^*$ of contention metrics through each port of the arbiter represented by t given requests r (cf. Equation 12.39 and Equation 12.40).

A metric of the type $Q_c : \dot{\mathbb{A}} \rightarrow (\{0, 1\}^* \rightarrow \mathbb{R})$ reducing the measure of contention to a point estimate as required by Equation 12.47 is derivable from X_2 by summing over the ports.

$$Q_c = \lambda t. \lambda r. \sum (X_2 t) r$$

E.2 Critical path length

A complementary time complexity metric to that of contention discussed in Section E.1 would estimate the latency of a request in the uncontended case by treating all primitive arbiters along a path as equal in cost. Such a metric would be made more descriptive by taking other primitives into account as well, with costs assessed either uniformly or based on the type primitive as a matter of discretion. Settling for unit costs on all components as in Section C.2 makes this problem a generalization of item 6c on page 409, although when not restricted to balanced crossbar arbiters it lacks a similarly appealing closed form and is solvable only up to a recurrence.

More complications follow from the dependence of the critical path in a token ring arbiter on the location of the token. A request to a port on a cell that does not hold the token entails a chain of events leading to the token-holding cell and back again before it can be granted. Because the token location is not known with certainty, the best we can hope to obtain is an average or expected critical path length rather than an exact figure, except for arbiters consisting exclusively of meshes and trees.

An exception to this exception pertains to the release acknowledgment phase in a token ring. Following the initial grant, there is no longer any uncertainty about the token location. The critical path during the release acknowledgment involves only the cell that issues the grant, and is fully determined by the components in it. Although helpfully informative, this effect nevertheless implies a requirement to keep track of a separate critical path for each phase.

E.2.1 Tree

Postponing these considerations for the moment, we can start with the easier case of a dendriform arbiter having $|u| - 1$ leaves such that the i -th leaf contains an arbiter with a critical path $u_{ij} \in \mathbb{R}$ through its j -th port for j ranging from 0 to $|u_i| - 1$, and $u_{|u_i|-1}$ similarly lists the critical paths through the root. The i -th leaf also contains a columnar decision wait of $|u_i|$ rows with the decomposition $d_i \in \mathbb{S}$ and critical paths specified by a function $T_g d_i : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ according to Equation C.11 as derived in Section C.2, or specifically the critical paths

$$l = (T_g d_i) \langle j, 0 \rangle \in \mathbb{R}^2 \tag{E.2}$$

for the j -th row output. Furthermore, the output MERGE network visible in Figure 12.4 has $|u_i|$ inputs, and hence a critical path of

$$\tau(|u_i|, j) \in \mathbb{R}$$

for the j -th input by Equation C.4, so there is a combined critical path of

$$\max \{l_0, \tau(|u_i|, j) + (u_{|d|})_i + l_1\}$$

covering the region from the FORK to the synchronization point implicit in the decision wait (excluding the FORK but including the decision wait) indicated in Figure 12.6. Note that l_0 is the path from the row input on the decision wait to the output, and l_1 is the path from the column input by Equation E.2 and Equation C.11 as illustrated in Figure C.2. The rest of the path highlighted in Figure 12.6 covers seven primitives including the FORK in addition to the path u_{ij} through the arbiter, for a total of

$$u_{ij} + 7 + \max \{l_0, \tau(|u_i|, j) + (u_{|d|})_i + l_1\} \in \mathbb{R}$$

through the j -th path of the i -th leaf,

$$(\lambda j. (\lambda l. u_{ij} + 7 + \max \{l_0, \tau(|u_i|, j) + (u_{|d|})_i + l_1\}) (T_g d_i) \langle j, 0 \rangle)^* \iota_{|u_i|} \in \mathbb{R}^{|u_i|}$$

for all paths through the i -th leaf, and $(Q_0 d) u \in \mathbb{R}^n$ for all paths through all leaves, with

$$Q_0 : \mathbb{S}^* \rightarrow (\mathbb{R}^{**} \rightarrow \mathbb{R}^*)$$

defined by

$$Q_0 = \lambda d. \lambda u. \flat (\lambda i. (\lambda j. (\lambda l. u_{ij} + 7 + \max \{l_0, \tau(|u_i|, j) + (u_{|d|})_i + l_1\}) (T_g d_i) \langle j, 0 \rangle)^* \iota_{|u_i|})^* \iota_{|d|}.$$

As noted above, the critical path length could differ between the initial request and the release acknowledgment phases. Although retracing these steps with respect to Figure 12.7 to find the latter phase critical path suggests an identical result, it may differ nevertheless to the extent u_{ij} varies between phases, for example due to the arbiter within the tree node being a token ring rather than a primitive. To allow for this possibility would require separate lists $u \in \mathbb{R}^{**}$ for each phase, or more succinctly a single list $v \in \mathbb{R}^{*2*}$ with $v_{i0} \in \mathbb{R}^*$ referring to the request phase and $v_{i1} \in \mathbb{R}^*$ to the release phase, so that $(Q_0 d)^* \langle v_0^T, v_1^T \rangle \in \mathbb{R}^{*2}$ expresses the result for each phase separately.

E.2.2 Mesh

Continuing under the assumption of a list $u \in \mathbb{R}^{**}$ specifying the critical path lengths $u_{ij} \in \mathbb{R}$ through the j -th port of the k -th arbiter, let a mesh be described by parameters $(b, m) \in \mathbb{N}^{**} \times \mathcal{P}(\mathbb{N})^{**}$ as proposed on page 370. We can group the lists of critical paths into stages by writing

$$u' = u \circ ((\lambda i. \iota_{|m_i|}^{\flat(m \cdot i)})^* \iota_{|m|})$$

(cf. Equation E.1) so that the critical path of the signal numbered s through the arbiter it traverses in i -th stage is

$$((h_1 u') (m, s)) i \in \mathbb{R}$$

by Equation 12.30, the list of critical paths through the arbiters in all $|m|$ stages for the signal numbered s is

$$((h_1 u') (m, s))^* \iota_{|m|} \in \mathbb{R}^{|m|}$$

and the list of all n such lists indexed by signal numbers is

$$q = (\lambda s. ((h_1 u') (m, s))^* \iota_{|m|})^* \iota_{\flat m_0} \in (\mathbb{R}^{|m|})^n$$

where $n = |b \ m_0|$ is the number of ports. (See Equation 11.2 for a reminder about this notation.) In the base case of $|m| = 1$ indicating a single stage, the mesh is restricted to either a wire or a primitive arbiter depending on whether $|m_{00}|$ is 1 or 2 respectively, with corresponding critical paths $q = \langle\langle 0 \rangle\rangle$ or $q = \langle\langle 1, \langle 1 \rangle \rangle$. In either case we may write $q = (Q_1 \ m) \ u$ for

$$Q_1 : \mathcal{P}(\mathbb{N})^{**} \rightarrow (\mathbb{R}^{**} \rightarrow \mathbb{R}^{**})$$

defined by

$$Q_1 = \lambda m. \lambda u. \langle (\lambda s. (h_1 (u \circ ((\lambda i. \iota_{|m_i|}^{b(m+i)}))^* \iota_{|m|}))) (m, s))^* \iota_{|m|} \rangle^* \iota_{|b \ m_0|}, \langle \delta_2^{|m_{00}|} \rangle_{\delta_1^{|m|}}^{\frac{|m_{00}|}{\delta_1^{|m|}}}. \quad (\text{E.3})$$

The mesh arbiter also incurs additional costs due to the paths through the broadcast network. Specifically the z -th zone for the s -th signal, which covers stages $l = \sum(b_s \ \vdash \ z)$ through $l + b_{sz} - 1$, includes a FORK network with b_s outputs and a JOIN network with b_s inputs. The total critical path for the s -th signal through i -th stage including the broadcast network is therefore

$$q_{si} + 2\tau(b_{sz}, i - l)$$

by Equation C.4 with $q = (Q_1 \ m) \ u$ by Equation E.3. Because the s -th signal traverses all b_{sz} paths through the z -th zone concurrently, the aggregate cost is

$$\max \mathcal{R}((\lambda l. (\lambda i. q_{si} + 2\tau(b_{sz}, i - l))^* \iota_{b_{sz}}^l) \sum(b_s \ \vdash \ z))$$

for that zone, and the cost of traversing all zones in sequence is

$$\sum_{z=0}^{|b_s|-1} \max \mathcal{R}((\lambda l. (\lambda i. q_{si} + 2\tau(b_{sz}, i - l))^* \iota_{b_{sz}}^l) \sum(b_s \ \vdash \ z)).$$

A list of all $n = |b|$ critical paths indexed by signal numbers follows as $(Q_2 \ b) \ q$ for

$$Q_2 : \mathbb{N}^{**} \rightarrow (\mathbb{R}^{**} \rightarrow \mathbb{R}^{**})$$

defined by

$$Q_2 = \lambda b. \lambda q. \left(\lambda s. \sum_{z=0}^{|b_s|-1} \max \mathcal{R}((\lambda l. (\lambda i. q_{si} + 2\tau(b_{sz}, i - l))^* \iota_{b_{sz}}^l) \sum(b_s \ \vdash \ z)) \right)^* \iota_{|b|}$$

or more explicitly $(Q_2 \ b) (Q_1 \ m) \ u$.

However, for a list $v \in \mathbb{R}^{*2*}$ with $v_{k0} \in \mathbb{R}^*$ describing the request phase critical paths of the k -th arbiter in the mesh, and v_{k1} describing the release phase critical paths, we would write

$$(Q_2 \ b)^* (Q_1 \ m)^* \langle\langle v_0^T, v_1^T \rangle, \langle \epsilon, \epsilon \rangle \rangle_{\delta_v} \in \mathbb{R}^{*2}$$

for the corresponding list of two lists of critical paths through the mesh arbiter each indexed by signal numbers. Because v_0^T and v_1^T are undefined if v is empty, the empty lists are noted explicitly as the arguments to $Q_1 \ m$ in this case, which coincides with the base case $|m| = 1$ for any valid mesh arbiter decomposition $((p, (b, m)), v) \in \mathcal{R}(\nabla_a)$ by Equation 12.24.

E.2.3 Token ring

For the token ring arbiter, we assume a list $w = \langle v_0^T, v_1^T \rangle \in \mathbb{R}^{**2}$ specifying the critical path $w_{0ij} \in \mathbb{R}$ during the request phase through the j -th of $|w_{0i}|$ ports on the arbiter in the i -th of $|w_0|$ cells both numbered from zero as usual, and an analogous interpretation for $w_{1ij} \in \mathbb{R}$ with respect to the release phase. We also take the list of functions

$$b \in (\{0, 1\}^2 \rightarrow \mathbb{R})^{|w_0|}$$

to specify the vertical critical paths through the 2-by-2 decision waits in the cells, with b_i pertaining to the i -th cell and $b_i \langle o_0, o_1 \rangle \in \mathbb{R}$ being the critical path from the input in column o_1 to the output in row o_0 and column o_1 (where column 0 is on the left as usual in Figure 12.9).

The analysis proceeds at first under the assumption of the highest numbered cell holding the token, and extends subsequently to unrestricted token distributions. Working somewhat backwards and from the inside out, we can identify a segment

$$b_i \langle 1, 1 \rangle + 2$$

in Figure 12.9 as part of the critical path from any request input on the cell numbered $i = |w_0| - 1$ to the corresponding grant output. This segment extends from the right column input on the 2-by-2 decision wait to the lower right output, then through a MERGE, a FORK and finally to the column input on the columnar decision wait. The part through $b_i \langle 1, 1 \rangle$ is traversed only in the last cell because only that cell holds the token and therefore has the lower row of its 2-by-2 decision wait enabled.

The alternative path from the right column input on the 2-by-2 decision wait in all other cells begins with $b_i \langle 0, 1 \rangle$, then goes through the SHUNT, the dotted output on TOGGLE below it, and the MERGE below that, out to the higher numbered cells toward the right, then back from the higher numbered cells through the SHUNT and the TOGGLE again, this time through the undotted output *en route* through the MERGE above the columnar decision wait and the FORK between them for a total of seven primitive component delays within the i -th cell. The path through the higher numbered cells is as follows.

- For each cell numbered l between the i -th cell and the last one, none of which holds the token, a signal arriving from the left traverses a MERGE, an arbiter, a TOGGLE, a 2-by-2 decision wait, and another MERGE to exit on the right. It then re-enters at the SHUNT, crosses another MERGE, then a FORK, and then repeats initial MERGE, arbiter, and TOGGLE sequence before exiting on the left. This path covers eight primitive component delays in addition to $b_l \langle 0, 0 \rangle$ through the decision wait and two trips through port 0 on the arbiter.
- For the last cell $l = |w_0| - 1$ only, the signal arriving from the left passes through the MERGE, arbiter, and TOGGLE, then through the 2-by-2 decision wait via $b_l \langle 1, 0 \rangle$, and then the MERGE, FORK, MERGE, arbiter, and TOGGLE sequence out to the left again, for a total of two trips through the arbiter, one through the decision wait, and six primitive component delays.

The two trips through the l -th arbiter incur a cost of $w_{0l0} + w_{1l0}$, the first term being its request phase critical path and the second its release phase on port 0. The total path length from the column input on the 2-by-2 decision wait to the column input on the columnar decision wait, including the seven primitives within the i -th cell and the rest in subsequent cells, therefore simplifies to

$$b_i \langle 0, 1 \rangle + 5 + \sum_{l=i+1}^{|w_0|-1} w_{0l0} + b_l \langle \delta_{|w_0|}^{l+1}, 0 \rangle + w_{1l0} + 8$$

for $0 \leq i < |w_0| - 1$, or to

$$\langle b_i \langle 0, 1 \rangle + 5 + \sum_{l=i+1}^{|w_0|-1} w_{0l0} + b_l \langle \delta_{|w_0|}^{l+1}, 0 \rangle + w_{1l0} + 8, b_i \langle 1, 1 \rangle + 2 \rangle_{\delta_{|w_0|}^{i+1}}$$

for $0 \leq i < |w_0|$ in general, which might be expressed more succinctly as $Q_3(b, w)_i$ in terms of a function

$$Q_3 : ((\{0, 1\}^2 \rightarrow \mathbb{R})^* \times \mathbb{R}^{**2}) \rightarrow \mathbb{R}^*$$

defined by

$$Q_3 = \lambda(b, w). \left(\lambda i. \langle b_i \langle 0, 1 \rangle + 5 + \sum_{l=i+1}^{|w_0|-1} w_{0l0} + b_l \langle \delta_{|w_0|}^{l+1}, 0 \rangle + w_{1l0} + 8, b_i \langle 1, 1 \rangle + 2 \rangle_{\delta_{|w_0|}^{i+1}} \right)^* \iota_{|w_0|}.$$

The segment $q_i = Q_3(b, w)_i$ is not the whole critical path through the i -th cell during the request phase. Along the rest, an external request first passes through the arbiter, a TOGGLE, and a FORK. One path beyond the FORK goes through a row input on the columnar decision wait, while the other passes through the multi-way MERGE, the segment q_i , and the column input on the columnar decision wait. Following the synchronization point between these two paths, there is one more MERGE to be traversed.

Before trying to express this idea formally, note that port 0 on each arbiter is not visible externally, so the j -th externally visible port on the i -th cell is associated with port $j + 1$ on the arbiter. Similarly, the multi-way MERGE has $|w_{0i}| - 1$ inputs rather than $|w_{0i}|$ for the arbiter, and the columnar decision wait has $|w_{0i}| - 1$ rows. To avoid any confusion, let $u_i = w_{0i} \ll 1 \in \mathbb{R}^{|w_{0i}|-1}$ denote the list of exposed critical paths through the i -th arbiter during the request phase, with

$$u_{ij} = (w_{0i})_{j+1} \in \mathbb{R}$$

referring internally to port $j + 1$. We also take the list

$$l \in \mathbb{R}^{2*|u|}$$

to describe the critical paths through the columnar decision waits such that $l_{ij0} \in \mathbb{R}$ is the path from the j -th row input on the columnar decision wait in the i -th cell to the j -th output on it, and l_{ij1} is from the column input. In these terms, the informal account above becomes

$$u_{ij} + 3 + \max \{ \tau(|u_i|, j) + q_i + l_{ij1}, l_{ij0} \} \in \mathbb{R}$$

for the j -th path through the i -th cell,

$$(\lambda j. u_{ij} + 3 + \max \{ \tau(|u_i|, j) + q_i + l_{ij1}, l_{ij0} \})^* \iota_{|u_i|} \in \mathbb{R}^{|u|}$$

for the list of all $|u_i|$ paths through the i -th cell, and $((Q_4 Q_3(b, w)) l) u \in \mathbb{R}^{*|u|}$ for the list of lists of paths through all $|u|$ cells, with

$$Q_4 : \mathbb{R}^* \rightarrow (\mathbb{R}^{2**} \rightarrow (\mathbb{R}^{**} \rightarrow \mathbb{R}^{**}))$$

defined by

$$Q_4 = \lambda q. \lambda l. \lambda u. (\lambda i. (\lambda j. u_{ij} + 3 + \max \{ \tau(|u_i|, j) + q_i + l_{ij1}, l_{ij0} \})^* \iota_{|u_i|})^* \iota_{|u|}. \quad (\text{E.4})$$

Accounting for the release phase critical paths is somewhat simpler because they do not pass through any decision waits or neighboring cells. After the i -th cell grants a request, the corresponding release signal to that cell takes a path that passes only through the arbiter, a TOGGLE, and a MERGE, for a result equal to the path through the arbiter plus two, with the arbiter critical paths given in this case by $w_{1i} \in \mathbb{R}^*$ according to the current convention. It is still necessary to note that the exposed ports exclude the first on each arbiter, but the expression

$$b (\lambda a. a + 2)^{**} (\lambda c. c \ll 1)^* w_1 \in \mathbb{R}^*$$

suffices for the list of all release phase critical paths, and it is not much more of an effort to express the critical paths for both phases as

$$b^* \langle (Q_4 Q_3(b, w)) l, (\lambda a. a + 2)^{**} \rangle \triangle (\lambda c. c \ll 1)^{**} w \in \mathbb{R}^{*2}$$

or more succinctly as $Q_5 \langle Q_3, Q_4 \rangle (l, b, w)$ with Q_5 defined by

$$Q_5 = \lambda q. \lambda(l, b, w). b^* \langle (q_1 q_0(b, w)) l, (\lambda a. a + 2)^{**} \rangle \triangle (\lambda c. c \ll 1)^{**} w. \quad (\text{E.5})$$

To express this result in terms of the decision wait decompositions $d \in \mathbb{S}^{*2}$ describing the token ring arbiter consistently with Equation 12.25, recall that $d_0 \in \mathbb{S}^*$ lists the columnar decision wait decompositions and $d_1 \in \mathbb{S}^{|d_0|}$ lists the 2-by-2 decision wait decompositions for all cells. By Equation C.11, the function $T_g d_{0i} : \mathbb{N}^2 \rightarrow \mathbb{R}^2$ specifies the critical paths for the columnar decision wait in the i -th cell, so the expression $(T_g d_{0i}) \langle j, 0 \rangle \in \mathbb{R}^2$ coincides with the interpretation of l_{ij} in Equation E.4, or more formally

$$l = (\lambda i. (\lambda j. (T_g d_{0i}) \langle j, 0 \rangle)^* \iota_{|w_{0i}|})^* \iota_{|d_0|}.$$

Similarly the function $T_g d_{1i} : \{0, 1\}^2 \rightarrow \mathbb{R}^2$ specifies the critical paths on the 2-by-2 decision wait in the i -th cell, with $(T_g d_{1i}) o \in \mathbb{R}^2$ being the paths to a given output at coordinates $o \in \{0, 1\}^2$, and specifically the path $((T_g d_{1i}) o)_1 \in \mathbb{R}$ being from the relevant column input to that output. A function

$$\lambda o. ((T_g d_{1i}) o)_1 : \{0, 1\}^2 \rightarrow \mathbb{R}$$

would map a given list $o \in \{0, 1\}^2$ to the columnar critical path through the 2-by-2 decision wait in the i -th cell, and the list of functions

$$b = (\lambda i. \lambda o. ((T_g d_{1i}) o)_1)^* \iota_{|d_1|}$$

would match the intended interpretation of the formal parameter b used in Equation E.5. Another way of writing the list of two lists of critical paths is then $(Q_6 Q_5 \langle Q_3, Q_4 \rangle) (d, w)$ with Q_6 defined by

$$Q_6 = \lambda q. \lambda(d, w). q((\lambda i. (\lambda j. (T_g d_{0i}) \langle j, 0 \rangle)^* \iota_{|w_{0i}|})^* \iota_{|d_0|}, (\lambda i. \lambda o. ((T_g d_{1i}) o)_1)^* \iota_{|d_1|}, w).$$

The last step in the derivation of token ring arbiter critical paths is to provide for general token distributions instead of assuming the token is held only in the highest numbered cell. If the token were in the i -th of $|k| = |d_0| = |w_0|$ cells instead (numbered from zero), and $s \in \mathbb{N}^{|k|}$ were a permutation that rotates a list of length $|k|$ by $i + 1$ positions to the left,

$$s = \iota_{|k|-i-1}^{i+1} \parallel \iota_{i+1}$$

then the critical path lengths would be obtained by rolling the ring until the i -th cell is in the last position, proceeding as above, and then rolling it back again

$$(q((d^T \circ s)^T, (w^T \circ s)^T)^T \circ s^{-1})^T \in \mathbb{R}^{*2}$$

given the same interpretations of d and w as above and $q = Q_6 Q_5 \langle Q_3, Q_4 \rangle$ (cf. Equation 12.37). If the token were only suspected of being in the i -th cell with probability k_i , then the expected critical path lengths would be the probability weighted sum

$$(\lambda e. \sum^* e^T)^* (\lambda i. k_i \cdot (q((d^T \circ s)^T, (w^T \circ s)^T)^T \circ s^{-1})^T)^* \iota_{|k|}^T \in \mathbb{R}^{*2}$$

which simplifies to

$$(\lambda e. \sum^* e^T)^* (\lambda i. k_i \cdot q((d^T \circ s)^T, (w^T \circ s)^T)^T \circ s^{-1})^* \iota_{|k|}$$

and is expressible more succinctly as $(Q_7 q) (d, k, w)$ with Q_7 defined by

$$Q_7 = \lambda q. \lambda(d, k, w). (\lambda e. \sum^* e^T)^* (\lambda i. k_i \cdot (\lambda s. q((d^T \circ s)^T, (w^T \circ s)^T)^T \circ s^{-1}) (\iota_{|k|-i-1}^{i+1} \parallel \iota_{i+1}))^* \iota_{|k|}.$$

E.2.4 General

A function $Q_8 : \mathbb{A} \rightarrow \mathbb{R}^{*2}$ taking any annotated decomposition $t = ((p, d, k), v) \in \mathbb{A}$ describing an arbiter to a list $Q_8 t \in \mathbb{R}^{*2}$ of lists of critical paths for each port during each of two phases follows mostly as a recurrence in terms of the functions Q_0 through Q_7 defined above subject to an additional provision for the permutation p .

$$Q_8 = \Lambda \lambda((p, d, k), v). (\lambda l. (l^T \circ p)^T) \begin{cases} (Q_0 d_0)^* \langle v_0^T, v_1^T \rangle & \text{if } d \in \mathbb{S}^{*1} \\ (Q_7 Q_6 Q_5 \langle Q_3, Q_4 \rangle) (d, k, \langle v_0^T, v_1^T \rangle) & \text{if } d \in \mathbb{S}^{*2} \\ (\lambda(b, m). (Q_2 b)^* (Q_1 m)^* \langle \langle v_0^T, v_1^T \rangle, \epsilon^z \rangle_{\delta_v}) d & \text{otherwise} \end{cases}$$

It might then be worthwhile to simplify this result to a single list $\sum^* (Q_8 t)^T \in \mathbb{R}^*$ by adding the paths along both phases for each port, which could be interpreted as the path through a complete cycle from an initial request to the final release acknowledgment. For a known request vector $r \in \{0, 1\}^*$, the projection

$$h = \langle \langle r, \sum^* (Q_8 t)^T \rangle^T \uparrow \langle 1 \rangle \parallel \mathbb{R}^1 \rangle_1^T \in \mathbb{R}^*$$

lists only the critical paths for the ports i on which requests are actually made (indicated by $r_i = 1$), and implies an “average” critical path length of

$$\frac{1}{|h|} \sum h$$

provided there is at least one request. A more meaningful average informed by known load and locality conditions would be obtained by plugging the related metric

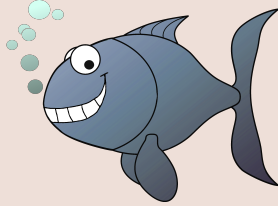
$$Q_{pa} = \lambda t. \lambda r. \left(\lambda h. \frac{\sum h}{\delta_0^{|h|} + |h|} \right) (\langle \langle r, \sum^* (Q_8 t)^T \rangle^T \uparrow \langle 1 \rangle \parallel \mathbb{R}^1 \rangle_1^T)$$

into Equation 12.47. Alternatively, sometimes the worst case critical path is more important than the average, which would be more like

$$Q_{pw} = \lambda t. \lambda r. \max \mathcal{R}(\langle \langle r, \sum^* (Q_8 t)^T \rangle^T \uparrow \langle 1 \rangle \parallel \mathbb{R}^1 \rangle_1^T) \quad (\text{E.6})$$

although plugging Q_{pw} into Equation 12.47 would still yield only the worst case on average.

Arbitrary exigencies

1. A crude estimate of latency is the square root of the number of components in a circuit based on the idea of a signal having to propagate a proportionate distance in a planar layout. What recurrence analogous to [Equation C.1](#) expresses the total number of components in an arbiter with a given decomposition? What shorter answer is there in terms of \mathcal{T}_{HL} ([Equation 8.23](#))?
- 
2. Define a measure of overall latency as a weighted sum of the contention and critical path length. How might this definition be modified to account for wire delays as well? (hint: [Appendix D](#)) How important would that be? (hint: [111, 274])
 3. Generalize the contention metric to allow a non-linear dependence on competing request probabilities in terms of tunable parameters that would be possible in principle to calibrate experimentally.
 4. Is there anything fishy about ignoring the PUSH and the horizontal critical paths through the 2-by-2 decision waits in the token ring arbiter, and if so, what should be done about it?
 5. What alternative to [Equation 12.47](#) in combination with [Equation E.6](#) could be used to estimate the worst of the worst case critical path lengths? What would it take to derive a distribution of critical path lengths and contention effects useful enough for a credible risk assessment?
 6. Solve [item 8](#) on page [409](#) and generalize the definitions of the contention and critical path metrics accordingly.

I never deny. I never contradict.
Sometimes I forget.

Benjamin Disraeli



DUAL RAIL BUFFER CELL THEORY OF OPERATION

Everything about the design of the buffer cell derived in [Section 14.1.3](#) flows from the idea of maintaining an invariant whereby the next output to come from the decision wait should originate from the left column whenever the cell stores a zero value, and from the right column otherwise. The PUSH to the left column input therefore starts the cell in a clear state. From there, the circuit is best understood by tracing each possible interaction with highlighter pens if necessary, starting with those that do not cause any change of state.

- If a read request arrives on r when the buffer holds zero ([Figure E5](#)), it goes straight through a MERGE and a SHUNT to the zero output data line \bar{o} .
- If a zero data input \bar{i} arrives when the cell already stores a zero ([Figure F1](#)), the upper left decision wait output sends a signal through the top MERGE and its output FORK, which signals the external acknowledgment a through one MERGE and maintains the zero state through the other MERGE and the PUSH.
- If a data input of one arrives via i when the buffer stores a value of one ([Figure F3](#)), the lower right output from the decision wait maintains the state by sending a signal through the MERGE below it and its output FORK back to the right column input, and emits an acknowledgment on a via the other output from the FORK and the MERGE in its path.

Otherwise we have three possible interactions that cause a change of state.

- If the buffer stores a value of zero and value of one is written to it via i ([Figure F2](#)), the lower left output from the decision wait sends a signal to the control input on the SHUNT below it, which passes through the dotted output from the TOGGLE below that and then back up through the MERGE and the FORK below the right column. The update of the state to one and the acknowledgment on a follow similarly from the FORK outputs to the case above.

- If a read request arrives on r when the buffer stores a value of one (Figure F.6), the following chain of events happens.
 - The signal from r goes through the MERGE and is shunted downward by the SHUNT to the TOGGLE directly below it, where it emerges this time through the undotted output.
 - The signal from the first TOGGLE goes through the data line of the next SHUNT in its path, then upward to the control line on the SHUNT above that one, and then down through the dotted output of the TOGGLE below it.
 - The signal from this TOGGLE goes through the MERGE connected to the top row of the decision wait, which generates a signal from the upper right output because the buffer stores a value of one.
 - The signal from the upper right output of the decision wait goes through the SHUNT again to be shunted downward through the TOGGLE below it, and to emerge this time from its undotted output.
 - The FORK following along this path sends an external output data signal of one via o , and also resets the state to zero by sending a signal through the MERGE and the PUSH connected to the left column input of the decision wait.
- If a signal to write zero arrives on \bar{i} when the buffer stores a value of one (Figure F.4), it has to clear the state and also reset the SHUNT near the r input, which entails the following chain of events.
 - The upper right output from the decision wait sends a signal through the data line on the SHUNT next to it, which proceeds through the control input on the SHUNT below it, and then through the dotted output of the TOGGLE below that.
 - The signal sent from this TOGGLE goes through the MERGE whose other input is r to the adjacent SHUNT, to be shunted downward through the undotted output of the TOGGLE below it. At this point the SHUNT and its adjacent TOGGLE have both been reset.
 - The output signal from this latter TOGGLE goes back through the previous SHUNT to be shunted downward and emerge from the undotted output of the TOGGLE below it.
 - The signal from this TOGGLE propagates all the way up to the top MERGE, and then down through the FORK below it.
 - This FORK sends one signal through a MERGE to the acknowledgment a , and another through the other adjacent MERGE through the PUSH to the left column input of the decision wait, thereby changing the state back to zero.

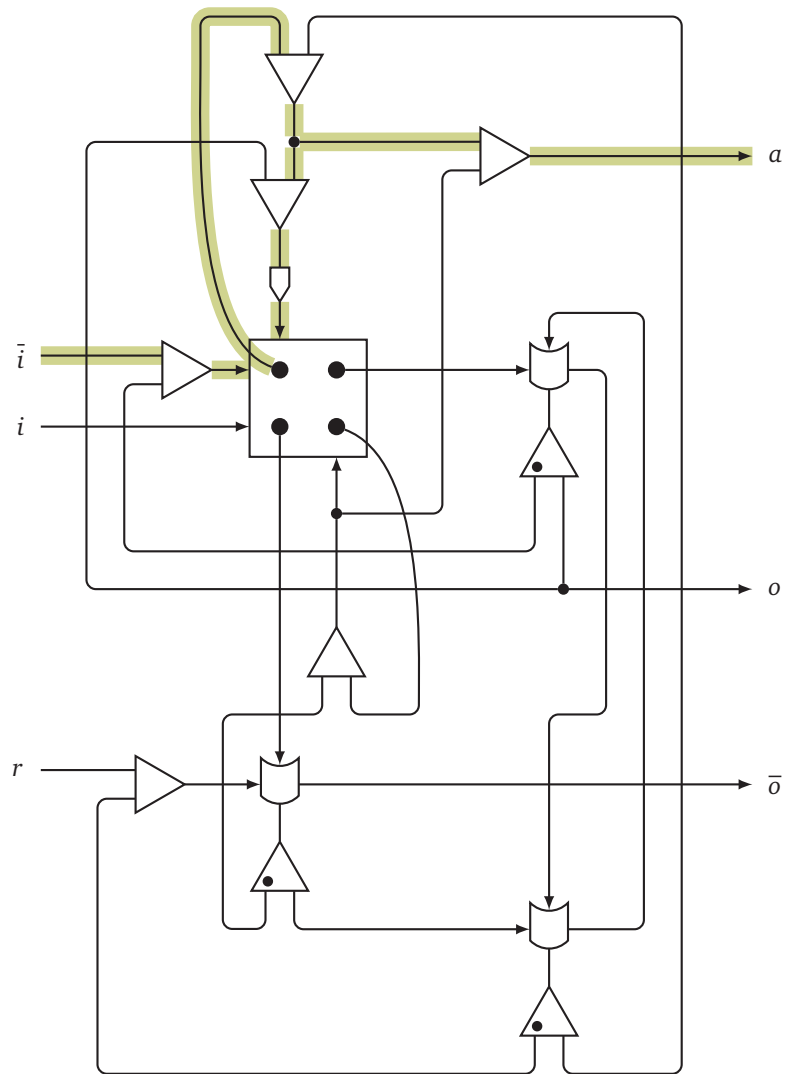


Figure F.1: Overwriting zero with zero maintains the left column of the decision wait as the next one to output.

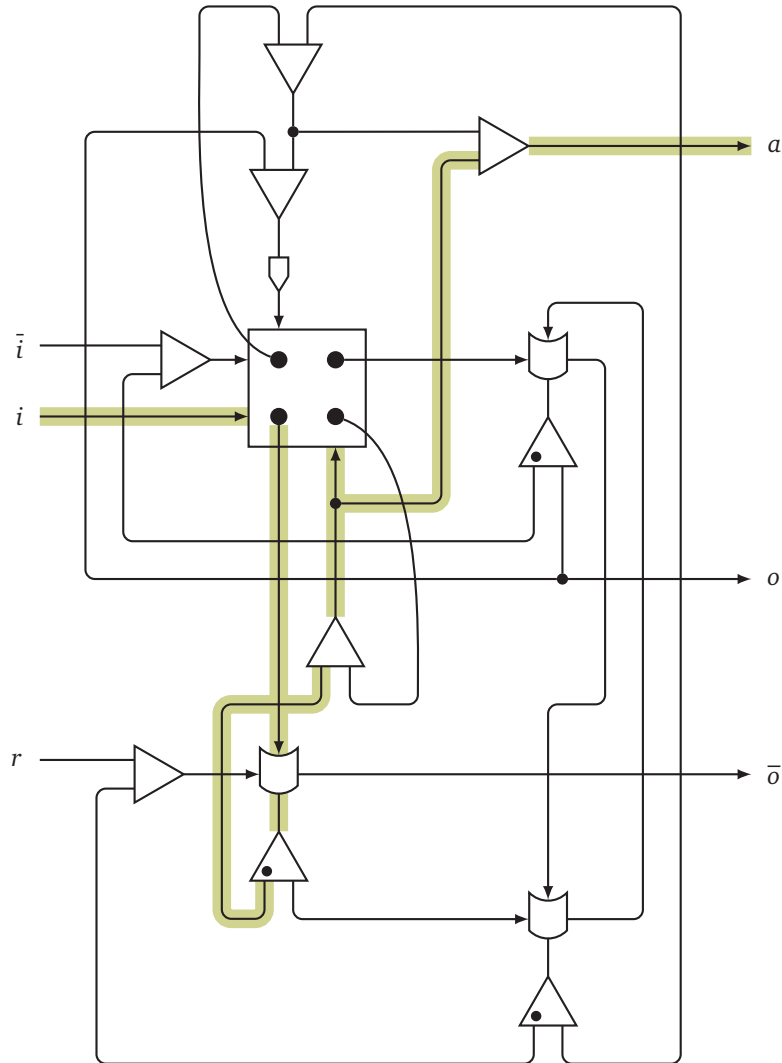


Figure F2: Overwriting zero with one uses the output from the left column of the decision wait to enable the right, and also prepares an alternative path for the read input through the SHUNT.

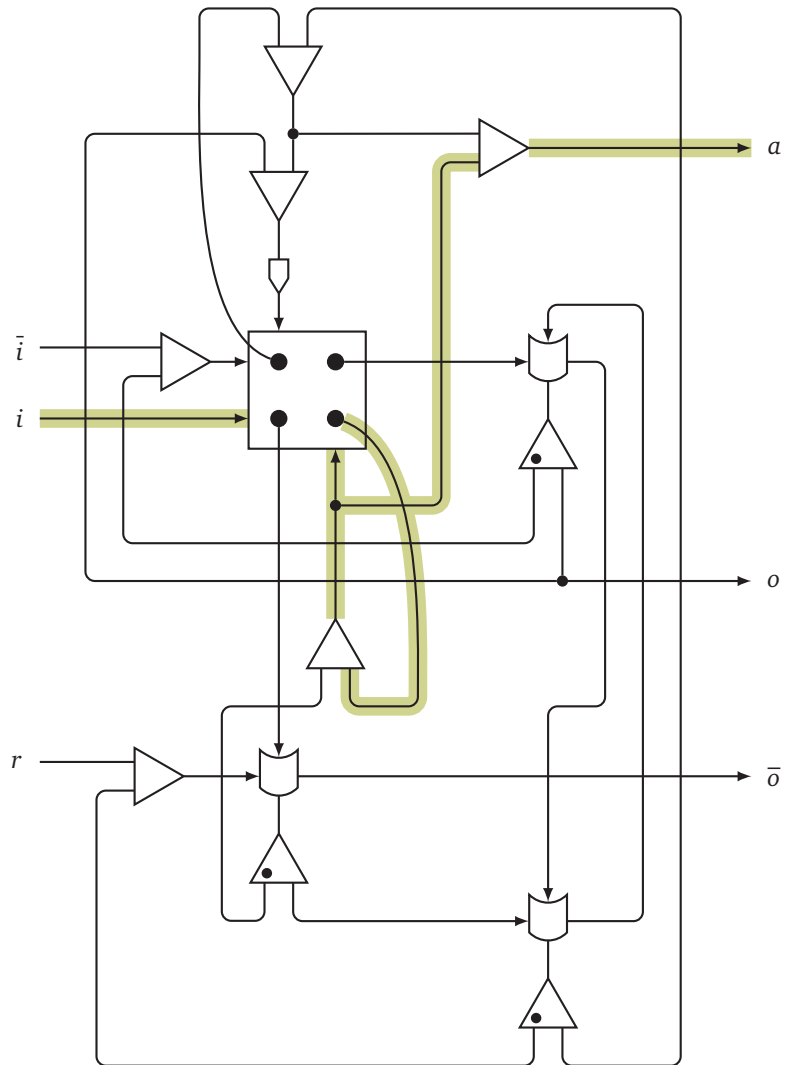


Figure F3: Overwriting one with one maintains the right column of the decision wait as the next one to output.

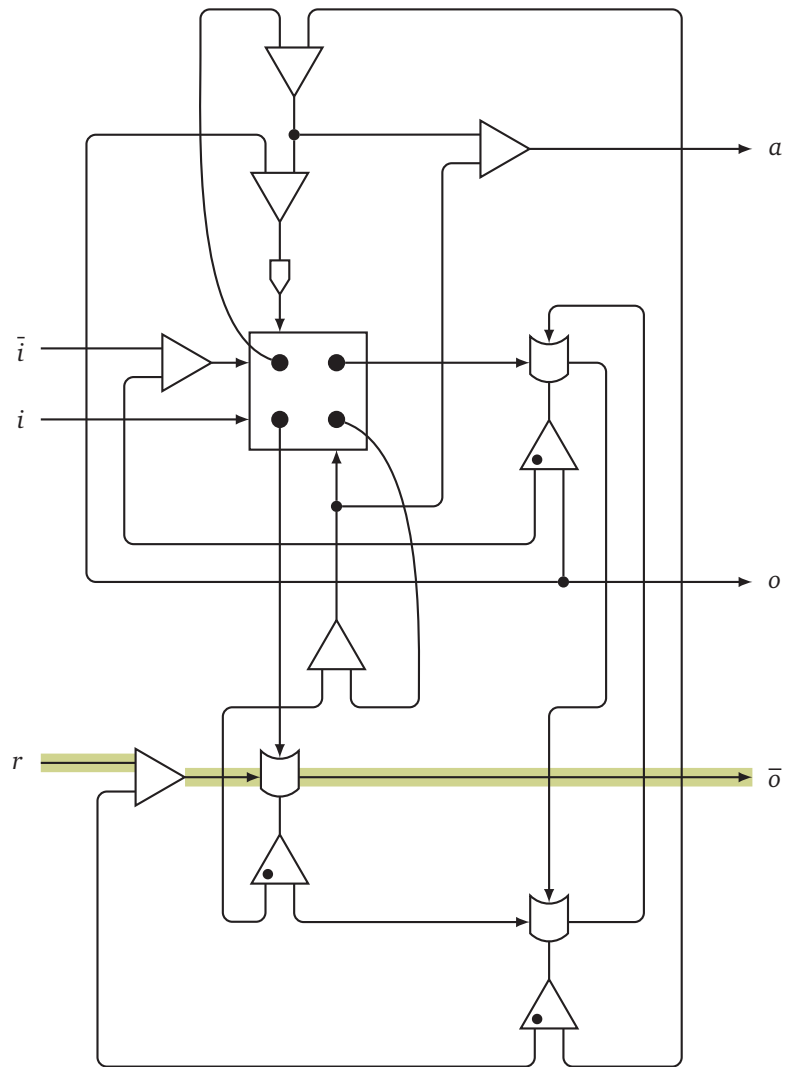


Figure F5: Reading zero is easy.

Buffer bother

1. What process combinator expression could be used to check whether the dual rail buffer cell design is correct according to [Equation 8.34](#)?
2. How should a FORK with a single exposed output be handled with regard to a critical path analysis?
3. How long is the critical path for each of the six interactions with
 - a) the paths internal to the decision wait neglected?
 - b) all relevant internal decision wait paths by [Equation 12.22](#) and [Equation C.11](#) considered?
4. Which of the eight possible decision wait decompositions minimizes the variation in critical path length?
5. What would it take to make the buffer cell store data persistently across multiple reads?
6. Design the persistent dual rail buffer cell proposed in the previous question and write its specification as a process combinator expression.



BIBLIOGRAPHY

- [1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–174. Springer, 2010. [cited on page [78](#), [80](#), [177](#)]
- [2] Samson Abramsky and Achim Jung. Domain theory. *Handbook of logic in computer science*, 3:1–168, 1994. [cited on page [579](#), [582](#)]
- [3] Samson Abramsky and C-H Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993. [cited on page [58](#)]
- [4] Susumu Adachi. Inner-independent radius-dependent totalistic rule of universal asynchronous cellular automaton. In *International Conference on Cellular Automata*, pages 546–555. Springer, 2014. [cited on page [252](#)]
- [5] Susumu Adachi, Jia Lee, Ferdinand Peper, and Hiroshi Umeo. Universality of 2-state asynchronous cellular automaton with inner-independent totalistic transitions. In *16th International Workshop on Cellular Automata and Discrete Complex Systems*, pages 153–172, 2010. [cited on page [252](#)]
- [6] Dharma P Agrawal. Graph theoretical analysis and design of multistage interconnection networks. *Computers, IEEE Transactions on*, 100(7):637–648, 1983. [cited on page [222](#)]
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, & Tools*. Addison-Wesley, 2006. [cited on page [55](#), [247](#), [270](#)]
- [8] Matthew An, J. Gregory Steffan, and Vaughn Betz. Speeding up FPGA placement: Parallel algorithms and methods. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 178–185. IEEE, 2014. [cited on page [21](#)]
- [9] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18.4:525–547, 2012. [cited on page [368](#)]
- [10] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Phil. Trans. R. Soc. A*, 375(2104), 2017. [cited on page [213](#)]
- [11] Markus Aronsson and Mary Sheeran. Hardware software co-design in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 162–173. ACM, 2017. [cited on page [20](#)]

- [12] Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985. [cited on page 213]
- [13] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013. [cited on page 21]
- [14] W. J. Bainbridge, William B. Toms, David A. Edwards, and Stephen B. Furber. Delay-insensitive, point-to-point interconnect using m-of-n codes. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 132–140. IEEE, 2003. [cited on page 45]
- [15] Zeineb Baklouti, David Duvivier, Rabie Ben Atitallah, Abdelhakim Artiba, and Nicolas Belanger. Real-Time Simulator supporting Heterogeneous CPU/FPGA Architecture. In *International Conference on Industrial Engineering and Systems Management*, Rabat, Maroc, October 2013. [cited on page 21]
- [16] Padmanabhan Balasubraminian and Nikos E. Mastorakis. Timing analysis of quasi-delay-insensitive ripple carry adders—a mathematical study. In *Proc. 3rd European Conference of Circuits Technology and Devices*, pages 233–240, 2012. [cited on page 575]
- [17] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000. [cited on page 253]
- [18] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Amsterdam, 1984. [cited on page 58]
- [19] Michael Barr and Charles Wells. *Category theory for computing science*, volume 1. Prentice Hall New York, 1990. [cited on page 123]
- [20] J. Beaumont. Variation tolerant asynchronous FPGA. Technical Report NCL-EEE-MICRO-TR-2018-208, Newcastle University, March 2018. [cited on page 18]
- [21] Peter A. Beerel, Georgios D. Dimou, and Andrew M. Lines. Proteus: An ASIC flow for GHz asynchronous designs. *IEEE Design & Test of Computers*, 28(5):36–51, 2011. [cited on page 18]
- [22] Václav E Beneš. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43(4):1641–1656, 1964. [cited on page 222]
- [23] Igor Benko and Jo Ebergen. Composing snippets. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 23–33, April 2000. [cited on page 534]
- [24] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992. [cited on page 32, 39, 575]
- [25] Eike Best, Raymond Devillers, and Maciej Koutny. The box algebra = Petri nets + process expressions. *Information and Computation*, 178(1):44 – 100, 2002. [cited on page 90]
- [26] Eike Best and Hans-Günther Linde-Göers. Compositional process semantics of Petri boxes. In *Mathematical Foundations of Programming Semantics*, pages 250–270. Springer, 1994. [cited on page 90]

- [27] Joseph K. Blitzstein and Jessica Hwang. *Introduction to Probability*. Texts in statistical science. CRC Press, 2014. [cited on page 363]
- [28] Gregor V. Bochmann. Hardware specification with temporal logic: An example. *Computers, IEEE Transactions on*, 100(3):223–231, 1982. [cited on page 258]
- [29] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, March 2003. [cited on page 535]
- [30] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999. [cited on page 360]
- [31] Gaetano Borriello, Carl Ebeling, Scott A. Hauck, and Steven Burns. The triptych FPGA architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):491–501, 1995. [cited on page 21]
- [32] Dominique Borrione, Menouer Boubekour, Laurent Mounier, Marc Renaudin, and Antoine Sirianni. Validation of asynchronous circuit specifications using IF/CADP. In *IFIP International Conference on VLSI SoC*, December 2003. [cited on page 41]
- [33] Gerald J. Brady, Austin J. Way, Nathaniel S. Safron, Harold T. Evensen, Padma Gopalan, and Michael S. Arnold. Quasi-ballistic carbon nanotube array transistors with current density exceeding Si and GaAs. *Science Advances*, 2(9), 2016. [cited on page 19]
- [34] C. Brey and Jim D. Garside. A quasi-delay-insensitive method to overcome transistor variation. In *VLSI Design, 2005. 18th International Conference on*, pages 368–373. IEEE, 2005. [cited on page 575]
- [35] Claude Brezinski and Michela Redivo Zaglia. *Extrapolation Methods Theory and Practice*, volume 2 of *Studies in Computational Mathematics*. North-Holland, 1991. [cited on page 400]
- [36] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, 1984. [cited on page 21]
- [37] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley Longman, Inc., 1995. [cited on page 20]
- [38] Stephen D. Brown and Zvonko G. Vranesic. *Fundamentals of digital logic with VHDL design*, volume 70125910. McGraw-Hill New York, 2000. [cited on page 18, 33, 36, 61, 450]
- [39] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. [cited on page 306]
- [40] Eric Brunvand. A community of asynchronauts: 20+ years of the ASYNC conference. In *This Asynchronous World, Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 22–58. Newcastle University, 2016. [cited on page 17]

- [41] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992. [cited on page 70]
- [42] J. A. Brzozowski. Delay-insensitivity and ternary simulation. *Theoretical Computer Science*, 2000. [cited on page 28, 36]
- [43] J. A. Brzozowski and H. Zhang. Delay-insensitivity and semi-modularity. Technical Report CS-97-11, Dept. of Comp. Science, Univ. of Waterloo, March 1997. [cited on page 36]
- [44] Janusz A. Brzozowski and Jo C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992. [cited on page 32]
- [45] Peter Buchholz and Peter Kemper. Hierarchical reachability graph generation for Petri nets. *Formal Methods in System Design*, 21(3):281–315, 2002. [cited on page 70]
- [46] Luca Cardelli, Marta Kwiatkowska, and Max Whitby. Chemical reaction network designs for asynchronous logic circuits. In *International Conference on DNA-Based Computers*, pages 67–81. Springer, 2016. [cited on page 20]
- [47] Josep Carmona, Jordi Cortadella, Victor Khomenko, and Alexandre Yakovlev. Synthesis of asynchronous hardware from Petri nets. In *Lectures on Concurrency and Petri Nets*, pages 345–401. Springer-Verlag, 2003. [cited on page 533]
- [48] Teena Carroll, David Galvin, and Prasad Tetali. Matchings and independent sets of a fixed size in regular graphs. *arXiv preprint arXiv:1206.3211v1*, November 2009. [cited on page 368]
- [49] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973. [cited on page 37, 407]
- [50] Thomas J. Chaney. My work on all things metastable OR me and my glitch. (self-published) last available at <https://arl.wustl.edu/~jst/cse/260/glitchChaney.pdf>. [cited on page 17, 37]
- [51] Thomas J. Chaney. Comments on “A note on synchronizer or interlock maloperation”. *IEEE Transactions on Computers*, (10):802–804, 1979. [cited on page 407]
- [52] Chihming Chang and Rami Melhem. Arbitrary size Benes networks. *Parallel Processing Letters*, 7(03):279–284, 1997. [cited on page 222]
- [53] Kai-Hui Chang, Valeria Bertacco, Igor L. Markov, and Alan Mishchenko. Logic synthesis and circuit customization using extensive external don’t-cares. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15(3):26, 2010. [cited on page 61]
- [54] Peter Y. K. Cheung. Are asynchronous ideas useful in FPGAs? In *This Asynchronous World, Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 87–95. Newcastle University, 2016. [cited on page 21]
- [55] Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. An algebraic theory of interface automata. Technical Report CS-RR-13-02, Department of Computer Science, University of Oxford, 2013. [cited on page 47, 75]

- [56] Gianfranco Ciardo. Reachability set generation for Petri nets: Can brute force be smart? In *Applications and Theory of Petri Nets 2004*, pages 17–34. Springer, 2004. [cited on page 70]
- [57] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Correct Hardware Design and Verification Methods*, pages 355–368. Springer, 2001. [cited on page 204]
- [58] Wesley A. Clark, Mishell J. Stucki, Severo M. Ornstein, and Charles E. Molnar. Macromodular computer design, part 1, volume 1, overview of macromodules. Technical Report 44, Computer Systems Laboratory, Washington University, February 1974. [cited on page 252]
- [59] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999. [cited on page 68]
- [60] Paulo Coelho. *The alchemist*. Harper Collins, 2007. [cited on page 41]
- [61] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34:171–210, 2002. [cited on page 20]
- [62] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H. Madden. Performance optimization of VLSI interconnect layout. *Integration, the VLSI journal*, 21(1):1–94, 1996. [cited on page 21]
- [63] Unicode Consortium. The unicode standard version 9.0.0. <http://www.unicode.org/versions/Unicode9.0.0/>, 2016. [cited on page 412]
- [64] James N. Cook. *Production rule verification for quasi-delay-insensitive circuits*. PhD thesis, California Institute of Technology, 1993. [cited on page 575]
- [65] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Third Edition*. MIT Press, 2009. [cited on page 324]
- [66] Krzysztof Czarnecki, John T. O’Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, pages 51–72. Springer, 2004. [cited on page 55]
- [67] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev. Moving from weakly endochronous systems to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science*, 146, 2006. [cited on page 17]
- [68] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998. [cited on page 17, 32]
- [69] Nicolaas Govert de Bruijn. Pólya’s theory of counting. *Applied combinatorial mathematics*, pages 144–184, 1964. [cited on page 356]
- [70] A. Prasanna de Silva and Nathan D. McClenaghan. Molecular-scale logic gates. *Chemistry-A European Journal*, 10(3):574–586, 2004. [cited on page 20]
- [71] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005. [cited on page 383]

- [72] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989. [cited on page 41, 47, 75, 77, 84, 100, 204, 258]
- [73] David L Dill and Edmund M Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings E (Computers and Digital Techniques)*, 133(5):276–282, 1986. [cited on page 258]
- [74] Charles Donnelly and Richard Stallman. *Bison: The YACC-compatible parser generator*. Free Software Foundation Cambridge (MA) 02139, 1992. [cited on page 55]
- [75] Luis Tarazona Duarte. *Performance-oriented syntax-directed synthesis of asynchronous circuits*. PhD thesis, University of Manchester, 2010. [cited on page 575]
- [76] Chris Dwyer, Moky Cheung, and Daniel J. Sorin. Semi-empirical SPICE models for carbon nanotube FET logic. In *In Proceedings of the Fourth IEEE Conference on Nanotechnology*, pages 35–39, 2004. [cited on page 19]
- [77] Matthew B Dwyer, Lori Clarke, et al. A compact Petri net representation and its implications for analysis. *Software Engineering, IEEE Transactions on*, 22(11):794–811, 1996. [cited on page 238]
- [78] Jo Ebergen and Robert Berks. VERDECT: A verifier for asynchronous circuits. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995. [cited on page 28]
- [79] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987. [cited on page 30, 32, 496]
- [80] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989. [cited on page 47]
- [81] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991. [cited on page 19, 575]
- [82] Jo C. Ebergen, John Segers, and Igor Benko. Parallel program and asynchronous circuit design. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design, Workshops in Computing*, pages 51–103. Springer-Verlag, 1995. [cited on page 252, 270]
- [83] Albert Einstein. *Relativity: The special and the general theory*. Princeton University Press, 2015. [cited on page 19]
- [84] Gavril F. Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3(3):261–73, 1973. [cited on page 368]
- [85] Karl M. Fant. *Logically Determined Design*. Wiley-Interscience. John Wiley & Sons, Inc., 2005. [cited on page 27]
- [86] Karl M. Fant and Scott A. Brandt. NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996. [cited on page 27, 253]

- [87] Stefan Felsner, Rudolf Miiller, and Lorenz Wernisch. Trapezoid graphs and generalizations, geometry and algorithms. *Discrete Applied Mathematics*, 74:13–32, 1997. [cited on page 368]
- [88] Thomas A. Feo, Mauricio G. C. Resende, and Stuart H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994. [cited on page 368]
- [89] M. Ferretti, R. Ozdag, and P. Beerel. High performance asynchronous ASIC back-end design flow using single-track full-buffer standard cells. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 95–105. IEEE Computer Society Press, April 2004. [cited on page 576]
- [90] Laurent Fesquet, Jerome Quartana, Marc Renaudin, et al. Asynchronous systems on programmable logic. In *ReCoSoC*, pages 105–112, 2005. [cited on page 21]
- [91] Mike Field. FPGA Webservice. https://github.com/hamsternz/FPGA_Webservice. [cited on page 20]
- [92] A. Flocke and T. G. Noll. Fundamental analysis of resistive nano-crossbars for the use in hybrid Nano/CMOS-memory. In *Proc. European Solid-State Circuits Conference (ESSCIRC)*, pages 328–331, September 2007. [cited on page 19]
- [93] Brendan Fong. *The algebra of open and interconnected systems*. PhD thesis, University of Oxford, 2016. [cited on page 205]
- [94] Dennis Furey. Efficient lattices for market calibrated derivatives valuation. Master’s thesis, Cass Business School, 2004. [cited on page 403]
- [95] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1979. [cited on page 360]
- [96] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal of Computing*, 1(2):180–187, June 1972. [cited on page 368]
- [97] James Gleick. *Chaos: Making a new science*. Random House, 1997. [cited on page 21]
- [98] Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996. [cited on page 68]
- [99] C. D. Godsil and M. W. Newman. Eigenvalue bounds for independent sets. *arXiv preprint arXiv:math/0508081v1*, 2005. [cited on page 368]
- [100] Alexander A. Green, Jongmin Kim, Duo Ma, Pamela A. Silver, James J. Collins, and Peng Yin. Complex cellular logic computation using ribocomputing devices. *Nature*, 2017. [cited on page 20]
- [101] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, 2009. [cited on page 363]

- [102] Carl A. Gunter, Peter D. Mosses, and Dana S. Scott. Semantic domains and denotational semantics. Technical Report MS-CIS-89-16, University of Pennsylvania, February 1989. [cited on page 579, 585]
- [103] John Michael Harris, Jeffrey L Hirst, and Michael J Mossinghoff. *Combinatorics and graph theory*, volume 2. Springer, 2008. [cited on page 356]
- [104] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995. [cited on page 19, 32]
- [105] Scott Hauck, Steven Burns, Geatano Borriello, and Carl Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994. [cited on page 21]
- [106] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985. [cited on page 21]
- [107] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. [cited on page 21]
- [108] A. Hertz and V. V. Lozin. The maximum independent set problem and augmenting graphs. In Avis D., Hertz A., and Marcotte O., editors, *Graph Theory and Combinatorial Optimization*. Springer, Boston, MA, USA, 2005. [cited on page 368]
- [109] Carl H. Heymann, Hendrik C. Ferreira, and Jos H. Weber. A Knuth-based RDS-minimizing multi-mode code. In *Information Theory Workshop (ITW), 2011 IEEE*, pages 508–512. IEEE, 2011. [cited on page 413]
- [110] J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and combinators: an introduction*, volume 13. Cambridge University Press Cambridge, 2008. [cited on page 58]
- [111] Ron Ho, Kenneth W. Mai, and Mark Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001. [cited on page 32, 619]
- [112] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice/Hall International, 1985. [cited on page 534]
- [113] Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, Jules Villard, Huibiao Zhu, and Peter O’Hearn. Developments in concurrent Kleene algebra. In *International Conference on Relational and Algebraic Methods in Computer Science*, pages 1–18. Springer, 2014. [cited on page 21]
- [114] Paul Hoffman. *The Man Who Loved Only Numbers: The Story of Paul Erdős and the Search for Mathematical Truth*. London: Fourth Estate, 1998. [cited on page 585]
- [115] John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001. [cited on page 21, 42, 70, 76, 78, 80, 149, 160, 178]
- [116] Heidi Howard and Richard Mortier. A generalised solution to distributed consensus. *arXiv preprint arXiv:1902.06776*, 2019. [cited on page 535]

- [117] Jiang Hu and Sachin S. Sapatnekar. A survey on multi-net global routing for integrated circuits. *Integration, the VLSI Journal*, 31(1):1–49, 2001. [cited on page 21]
- [118] David A. Huffman. The design and use of hazard-free switching networks. *J. ACM*, 4(1):47–62, January 1957. [cited on page 23]
- [119] N. Huot, H. Dubreuil, Laurent Fesquet, and Marc Renaudin. FPGA Architecture for Multi-Style Asynchronous Logic. In *Design, Automation, and Test in Europe*, pages 32–33, 2005. [cited on page 21]
- [120] Masashi Imai, Tomohiro Yoneda, and Takashi Nanya. N-way ring and square arbiters. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 125–130. IEEE, 2009. [cited on page 361, 364]
- [121] Nabil Imam and Rajit Manohar. Address-event communication using token-ring mutual exclusion. In *Asynchronous Circuits and Systems (ASYNC), 2011 17th IEEE International Symposium on*, pages 99–108. IEEE, 2011. [cited on page 383]
- [122] Kees A. Schouhamer Immink. *Codes for mass data storage systems*. Shannon Foundation Publisher, 2004. [cited on page 413]
- [123] Lubomir Ivanov and Ramakrishna Nunna. Modeling and verification of cache coherence protocols. In *IEEE International Symposium on Circuits and Systems*, pages 129–132, 2001. [cited on page 21]
- [124] K. W. James and K. Y. Yun. Average-case optimized transistor-level technology mapping of extended burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 70–79, 1998. [cited on page 36]
- [125] Steven Dexter Johnson and Bhaskar Bose. DDD: A system for mechanized digital design derivation. Technical Report 323, Computer Science Department, Indiana University, 1990. [cited on page 204]
- [126] Geraint Jones. Designing circuits by calculation. Technical Report PRG TR-10-90, Programming Research Group, Oxford University Computing Laboratory, 1990. [cited on page 204]
- [127] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In *3rd Refinement Workshop*, volume 90, pages 133–152, 1990. [cited on page 204]
- [128] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992. [cited on page 47, 77, 100]
- [129] Mark B. Josephs and Hemangee K. Kapoor. Controllable delay-insensitive processes. *Fundamenta Informaticae*, 78(1):101–130, 2007. [cited on page 572]
- [130] Stasys Jukna. *Extremal Combinatorics With Applications in Computer Science*. Texts in Theoretical Computer Science. Springer-Verlag, 2nd edition, 2011. [cited on page 413, 414]
- [131] Hiroto Kagotani and Takashi Nanya. Synthesis of two-phase quasi-delay-insensitive circuits from dependency graphs. *Systems and computers in Japan*, 26(4):11–19, 1995. [cited on page 575]

- [132] Rajgopal Kannan. The KR-Benes network: a control-optimal rearrangeable permutation network. *Computers, IEEE Transactions on*, 54(5):534–544, 2005. [cited on page 222]
- [133] Hemangee K. Kapoor, Mark B. Josephs, and Dennis Furey. Verification and implementation of delay-insensitive processes in restrictive environments. *Fundamenta Informaticae*, 70(1):21–48, 2006. [cited on page 61, 572]
- [134] R. Karmazin, C.T. Ortega Otero, and R. Manohar. CellTK: Automated layout for asynchronous circuits with nonstandard cells. In *The 19th International Symposium on Asynchronous Circuits and Systems*. IEEE, 2013. [cited on page 18]
- [135] Randy H. Katz and Gaetano Borriello. *Contemporary Logic Design*. Pearson Prentice Hall, 2005. [cited on page 450]
- [136] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974. [cited on page 19, 28, 252, 260, 575]
- [137] Robert M. Keller. Computer science: Abstraction to implementation. (self-published) last available at <https://www.cs.hmc.edu/~keller/cs60book/%20%20%20All.pdf>, 2001. [cited on page 19]
- [138] Sean Keller, Michael Katelman, and Alain J. Martin. A necessary and sufficient timing assumption for speed-independent circuits. In *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*, pages 65–76. IEEE, 2009. [cited on page 32]
- [139] Jeremy Kepner and John Gilbert (eds.). *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, and Tools. SIAM, 2011. [cited on page 324]
- [140] David J. Kinniment. He who hesitates is lost. (self-published) last available at <http://www.async.org.uk/David.Kinniment/DJKinniment-He-Who-Hesitates-is-Lost.pdf>. [cited on page 17]
- [141] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986. [cited on page 213]
- [142] John C. Knight and Nancy G. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, 15(1):24–35, 1990. [cited on page 213]
- [143] Donald E. Knuth. *The Art of Computer Programming, Volume 3 / Sorting and Searching*. Addison-Wesley, 1973. [cited on page 18]
- [144] Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. <https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf>, March 1977. [cited on page 213]
- [145] Donald E. Knuth. Efficient balanced codes. *IEEE Transactions on Information Theory*, 32(1):51–53, 1986. [cited on page 413]
- [146] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005. [cited on page 367]

- [147] Cheng-Kok Koh and Patrick H. Madden. Manhattan or non-Manhattan?: a study of alternative VLSI routing architectures. In *Proceedings of the 10th Great Lakes symposium on VLSI*, pages 47–52. ACM, 2000. [cited on page 21]
- [148] Alex Kondratyev, Michael Kishinevsky, Bill Lin, Peter Vanbekbergen, and Alex Yakovlev. Basic gate implementation of speed-independent circuits. In *Design Automation, 1994. 31st Conference on*, pages 56–62. IEEE, 1994. [cited on page 32]
- [149] Alex Kondratyev, Michael Kishinevsky, and Alexandre Yakovlev. Hazard-free implementation of speed-independent circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(9):749–771, 1998. [cited on page 32]
- [150] Thomas S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012. [cited on page 577]
- [151] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008. [cited on page 20]
- [152] Michihiro Kuramochi and George Karypis. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 313–320. IEEE, November 2001. [cited on page 534]
- [153] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005. [cited on page 535]
- [154] Joseph Lauer and Nicholas Wormald. Large independent sets in regular graphs of large girth. *Journal of Combinatorial Theory, Series B*, 97:999–1009, 2007. [cited on page 368]
- [155] Jakob Lechner, Andreas Steininger, and Florian Huemer. Methods for analysing and improving the fault resilience of delay-insensitive codes. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 519–526. IEEE, 2015. [cited on page 415]
- [156] Edward A. Lee and Thomas Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995. [cited on page 21]
- [157] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srinivas Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176–179. IEEE, 2004. [cited on page 441]
- [158] Hyung Lee-Kwang, Joël Favrel, and Pierre Baptiste. Generalized Petri net reduction method. *Systems, Man and Cybernetics, IEEE Transactions on*, 17(2):297–303, 1987. [cited on page 68, 238]
- [159] Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. Mathematics for computer science. <https://courses.csail.mit.edu/6.042/spring18/mcs.pdf>, 2018. [cited on page 18, 363]
- [160] Guy G. Lemieux and Stephen D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *ACM-SIGDA Physical Design Workshop*, 1993. [cited on page 21]

- [161] Jacques Lenfant. Parallel permutations of data: A Benes network control algorithm for frequently used permutations. *Computers, IEEE Transactions on*, 100(7):637–647, 1978. [cited on page 222]
- [162] Gavriela Freund Lev, Nicholas Pippenger, and Leslie G Valiant. A fast parallel algorithm for routing in permutation networks. *Computers, IEEE Transactions on*, 100(2):93–100, 1981. [cited on page 222]
- [163] Oscar Levin. Discrete mathematics, an open introduction. <http://discretetext.oscarlevin.com/pdfs/dmoi-tablet.pdf>, 2016. [cited on page 18, 88, 123]
- [164] Jens Lienig. A parallel genetic algorithm for performance-driven VLSI routing. *Evolutionary Computation, IEEE Transactions on*, 1(1):29–39, 1997. [cited on page 21]
- [165] Jens Lienig and Krishnaiyan Thulasiraman. A genetic algorithm for channel routing in VLSI circuits. *Evolutionary Computation*, 1(4):293–311, 1993. [cited on page 21]
- [166] Francisco S. N. Lobo. Exotic solutions in general relativity: Traversable wormholes and “warp drive” spacetimes. *arXiv preprint arXiv:0710.4474*, 2007. [cited on page 32]
- [167] Hock Soon Low, Delong Shang, Fei Xia, and Alex Yakovlev. Variation tolerant asynchronous FPGA. Technical Report NCL-EECE-MSD-TR-2010-163, Newcastle University, December 2010. [cited on page 21]
- [168] Mei Lu, Huiqing Liu, and Feng Tian. Laplacian spectral bounds for clique and independence numbers of graphs. *Journal of Combinatorial Theory, Series B*, 97:726–732, 2007. [cited on page 368]
- [169] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. *Systolic array processors*, pages 589–598, 1989. [cited on page 204]
- [170] Carlos J. Luz. An upper bound on the independence number of a graph computable in polynomial-time. *Operations Research Letters*, 18:139–145, 1995. [cited on page 368]
- [171] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, 1987. [cited on page 47]
- [172] W. C. Mallon, J. T. Udding, and T. Verhoeff. Analysis and applications of the XDI model. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 231–242, April 1999. [cited on page 36, 572]
- [173] B. B. Mandelbrot. *The Fractal Geometry of Nature*. Einaudi paperbacks. Henry Holt and Company, 1983. [cited on page 21]
- [174] Reinhard Männer. Metastable states in asynchronous digital systems: Avoidable or unavoidable? *Microelectronics Reliability*, 28(2):295–307, 1988. [cited on page 407]
- [175] M. Morris Mano, Charles R. Kime, and Tom Martin. *Logic and Computer Design Fundamentals*. Pearson Higher Education, Inc., 2015. [cited on page 361, 450]

- [176] R. Manohar and A. J. Martin. Quasi-delay-insensitive circuits are Turing complete. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996. [cited on page 575]
- [177] Rajit Manohar. Reconfigurable asynchronous logic. In *IEEE Custom Integrated Circuits Conference*, pages 13–20, 2006. [cited on page 21]
- [178] Rajit Manohar and Yoram Moses. Analyzing isochronic forks with potential causality. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, May 2015. [cited on page 32]
- [179] Alain J. Martin. A delay-insensitive fair arbiter. Technical Report 5193:TR:85, California Institute of Technology, 1985. [cited on page 361]
- [180] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985. [cited on page 361]
- [181] Alain J. Martin. On Seitz’s arbiter. Technical Report 5212:TR:86, California Institute of Technology, March 1985. [cited on page 258]
- [182] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990. [cited on page 32, 577]
- [183] Alain J. Martin. 25 years ago: The first asynchronous microprocessor. Technical Report CS-TR-1-2014, California Institute of Technology, January 2014. [cited on page 575]
- [184] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI*, pages 351–373. MIT Press, 1989. [cited on page 575]
- [185] Alain J. Martin and Mika Nystrom. Asynchronous techniques for system-on-chip design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006. [cited on page 575]
- [186] Alain J. Martin, Mika Nystrom, and Catherine G. Wong. Three generations of asynchronous microprocessors. *IEEE Design & Test of Computers*, 20(6):9–17, 2003. [cited on page 575]
- [187] Pavlos M. Mattheakis and Christos P. Sotiriou. Polynomial complexity asynchronous control circuit synthesis of concurrent specifications based on burst-mode FSM decomposition. In *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, pages 251–256, 2013. [cited on page 18]
- [188] Arya Mazumdar, Ron M. Roth, and Pascal O. Vontobel. On linear balancing sets. In *Information Theory, 2009. ISIT 2009. IEEE International Symposium on*, pages 2699–2703. IEEE, 2009. [cited on page 415]
- [189] Gordon McComb et al. *Electronics for dummies*. John Wiley & Sons, 2011. [cited on page 219]
- [190] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, pages 73–88. Springer, 2004. [cited on page 55]

- [191] Michael Mendler and Terry Stroup. Newtonian arbiters cannot be proven correct. *Formal Methods in System Design*, 3(3), December 1993. [cited on page 258, 407, 609]
- [192] Chris Meyers. Asynchronous design and beyond. In *This Asynchronous World, Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 236–240. Newcastle University, 2016. [cited on page 18]
- [193] Dimitrios Milios. Probability distributions as program variables. Master’s thesis, School of Informatics, University of Edinburgh, 2009. [cited on page 605]
- [194] Gabriele Miorandi, Davide Bertozzi, and Steven M. Nowick. Increasing impartiality and robustness in high-performance n-way asynchronous arbiters. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 108–115. IEEE, 2015. [cited on page 361, 365]
- [195] Ian Mitchell. Proving Newtonian arbiters correct, almost surely. Master’s thesis, The University of British Columbia, October 1994. [cited on page 609]
- [196] Isi Mitrani and Alex Yakovlev. Tree arbiter with nearest-neighbour scheduling. In *Advances in Computer and Information Sciences*. ISCIS, 1998. [cited on page 409]
- [197] Ethan Mollick. Establishing Moore’s law. *Annals of the History of Computing, IEEE*, 28(3):62–75, 2006. [cited on page 19]
- [198] Eric Monmasson and Marciam N. Cirstea. FPGA design methodology for industrial control systems – A review. *IEEE Transactions on Industrial Electronics*, 54(4), 2007. [cited on page 21]
- [199] Michael S. Morris, Kip S. Thorne, and Ulvi Yurtsever. Wormholes, time machines, and the weak energy condition. *Physical Review Letters*, 61(13):1446, 1988. [cited on page 32]
- [200] Daniel Morrison and Irek Ulidowski. Arbitration and reversibility of parallel delay-insensitive modules. In *International Conference on Reversible Computation*, pages 67–81. Springer, 2014. [cited on page 253]
- [201] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77:541–580, 1989. [cited on page 21, 49]
- [202] John Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, October 1984. [cited on page 531]
- [203] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994. [cited on page 575]
- [204] Syed Rameez Naqvi and Andreas Steininger. A tree arbiter cell for high speed resource sharing in asynchronous environments. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 295. European Design and Automation Association, 2014. [cited on page 361]
- [205] Nam-phuong D. Nguyen, Hiroyuki Kuwahara, Chris J. Myers, and James P. Keener. The design of a genetic Muller C-element. In *The 13th IEEE International Symposium on Asynchronous Circuits and Systems*, 2007. [cited on page 20]

- [206] Jean P. Nicolle. Where FPGAs are fun. <http://www.fpga4fun.com/>. [cited on page 20]
- [207] Rishiyur S. Nikhil. *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*, pages 129–146. Springer Netherlands, Dordrecht, 2008. [cited on page 23]
- [208] S. Nikolettseas, C. Raptopoulos, and P. Spirakis. Large independent sets in general random intersection graphs. *Theoretical Computer Science*, 406(3):215–224, October 2008. [cited on page 368]
- [209] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance*, pages 9–16. ACM, 2013. [cited on page 55]
- [210] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993. [cited on page 32, 36]
- [211] Richard K. Obousy and Eric W. Davis. Warp drive, dark energy, and the manipulation of extra dimensions. Technical Report DIA-08-1004-001, Defense Intelligence Agency, December 2009. [cited on page 32]
- [212] John o'Donnell. Teaching functional circuit specification in hydra. In *Functional Programming Languages in Education*, pages 195–214. Springer, 1995. [cited on page 204]
- [213] Severo M. Ornstein, Mishell J. Stucki, and Wesley A. Clark. A functional description of macromodules. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 337–355, Atlantic City, NJ, 1967. Academic Press. [cited on page 252]
- [214] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*, volume 1. Prentice Hall Professional, 2003. [cited on page 23]
- [215] André Pang, Don Stewart, Sean Seefried, and Manuel MT Chakravarty. Plugging Haskell in. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM, 2004. [cited on page 55]
- [216] Hongsik Park, Ali Afzali, Shu-Jen Han, George S Tulevski, Aaron D Franklin, Jerry Tersoff, James B Hannon, and Wilfried Haensch. High-density integration of carbon nanotubes via chemical self-assembly. *Nature nanotechnology*, 7(12):787–791, 2012. [cited on page 19]
- [217] Terence Parr et al. Antlr parser generator. *Online Stand Dezember*, 2009. [cited on page 55]
- [218] Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998. [cited on page 32]
- [219] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, June 1994. [cited on page 68, 238]
- [220] Priyadarsan Patra. *Approaches to Design of Circuits for Low-Power Computation*. PhD thesis, The University of Texas at Austin, 1995. [cited on page 285]

- [221] Priyadarsan Patra and Donald Fussell. Efficient building blocks for delay insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 196–205, November 1994. [cited on page 74, 575]
- [222] Priyadarsan Patra and Donald S. Fussell. Building-blocks for designing DI circuits. Technical report tr93-23, Dept. of Computer Sciences, The University of Texas at Austin, November 1993. [cited on page 252, 253, 254, 260, 263, 277, 278, 421]
- [223] Priyadarsan Patra and Donald S. Fussell. Fully asynchronous, robust, high-throughput arithmetic structures. In *Proc. of Eighth International Conference on VLSI Design*. IEEE Computer Society Press, January 1995. [cited on page 19, 412, 575]
- [224] Priyadarsan Patra, Donald S. Fussell, and Stanislav Polonsky. Delay insensitive logic for RSFQ superconductor technology. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 42–53. IEEE Computer Society Press, April 1997. [cited on page 20]
- [225] Volnei A. Pedroni. *Circuit design with VHDL*. MIT press, 2004. [cited on page 23]
- [226] Ad Peeters and Kees van Berkel. Single-rail handshake circuits. In *Asynchronous Design Methodologies*, pages 53–62. IEEE Computer Society Press, May 1995. [cited on page 253]
- [227] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993. [cited on page 68]
- [228] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated synthesis for asynchronous FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 163–173. ACM, 2005. [cited on page 21]
- [229] Ferdinand Peper, Jia Lee, Susumu Adachi, and Shinro Mashiko. Laying out circuits on asynchronous cellular arrays: a step towards feasible nanocomputers? *Nanotechnology*, 14:469–485, 2003. [cited on page 19]
- [230] Stanislaw J. Piestrak and Takashi Nanya. Towards totally self-checking delay-insensitive systems. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 228–237. IEEE, 1995. [cited on page 575]
- [231] Juha Plosila, R. Rukšenas, and Kaisa Sere. Delay-insensitive circuits and action systems. Technical Report 60, Turku Centre for Computer Science, November 1996. [cited on page 575]
- [232] Ivan Poliakov. *Interpreted Graph Models*. PhD thesis, Schoole of EECE, Newcastle University, 2011. [cited on page 41]
- [233] Juan Pontes, Ney Calazans, and Pascal Vivet. H2A: A hardened asynchronous network on chip. In *Integrated Circuits and Systems Design (SBCCI), 2013 26th Symposium on*, pages 1–6. IEEE, 2013. [cited on page 575]
- [234] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014. [cited on page 21]

- [235] Julian Rathke, Paweł Sobociński, and Owen Stephens. Compositional reachability in Petri nets. In *Reachability Problems*, pages 230–243. Springer, 2014. [cited on page 90]
- [236] Raúl Rojas. A tutorial introduction to the lambda calculus. *arXiv preprint arXiv:1503.09060*, 2015. [cited on page 58]
- [237] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall (Pearson), 2005. [cited on page 534]
- [238] Saleh Safiruddin. Single electron tunneling based building blocks for delay insensitive circuits. Master’s thesis, Delft University of Technology, 2008. [cited on page 20]
- [239] Maarten P D. Schadd, Mark H. M. Winands, H. Jaap Van Den Herik, Guillaume M. J-B. Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 1–12. Springer, 2008. [cited on page 306]
- [240] David A. Schmidt. Denotational semantics: A methodology for language development. (self-published) last available at <http://people.cs.ksu.edu/~schmidt/text/DenSem-full-book.pdf>, 1997. [cited on page 579]
- [241] Karsteb Schmidt. Stubborn sets for standard properties. In *Application and Theory of Petri Nets 1999*, pages 46–65. Springer, 1999. [cited on page 68]
- [242] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PR6-6, Oxford University Computing Laboratory, August 1971. [cited on page 579]
- [243] Roberto Segala. Quiescence, fairness, testing, and the notion of implementation. In *CONCUR’93*, pages 324–338. Springer, 1993. [cited on page 75]
- [244] J. P. L. Segers. The design and analysis of asynchronous up-down counters. Master’s thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, June 1993. [cited on page 252, 270]
- [245] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980. [cited on page 361]
- [246] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980. [cited on page 361]
- [247] Jakov N Seizovic. Pipeline synchronization. In *Advanced Research in Asynchronous Circuits and Systems, 1994., Proceedings of the International Symposium on*, pages 87–96. IEEE, 1994. [cited on page 258]
- [248] John M. Shalf and Robert Leland. Computing beyond moore’s law. *Computer*, 48(12):14–23, 2015. [cited on page 19]
- [249] Maitham Shams. *Modeling and Optimization of CMOS Logic Circuits with Application to Asynchronous Design*. PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, May 1999. [cited on page 252]
- [250] Maitham Shams, Jo C. Ebergen, and Mohamed I. Elmasry. Modeling and comparing CMOS implementations of the C-element. *IEEE Transactions on VLSI Systems*, 6(4):563–567, December 1998. [cited on page 29]

- [251] Naresh R Shanbhag, Subhasish Mitra, Gustavode de Veciana, Michael Orshansky, Radu Marculescu, Jaijeet Roychowdhury, Douglas Jones, and Jan M Rabaey. The search for alternative computational paradigms. *IEEE Design & Test of Computers*, (4):334–343, 2008. [cited on page 19]
- [252] Robin Sharp and Ole Rasmussen. Using a language of functions and relations for VLSI specification. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 45–54. ACM, 1995. [cited on page 204]
- [253] Sol M. Shatz, Shengru Tu, Tadao Murata, and Sastry Duri. An application of Petri net reduction for Ada tasking deadlock analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 7(12):1307–1322, 1996. [cited on page 68, 238]
- [254] Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM, 1984. [cited on page 204]
- [255] Mary Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158, 2005. [cited on page 204]
- [256] Yebin Shi, Steve B. Furber, Jim Garside, and Luis A. Plana. Fault tolerant delay insensitive inter-chip communication. In *The 15th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 77–84. IEEE, 2009. [cited on page 415]
- [257] Klaus Simon. A note on lexicographic breadth first search for chordal graphs. *Information Processing Letters*, 54:249–251, 1995. [cited on page 368]
- [258] T. Singh and A. Taubin. A highly scalable GALS crossbar using token ring arbitration. *IEEE Design & Test of Computers*, 24:464–472, September 2007. [cited on page 383]
- [259] Kostas Siozios, George Koutroumpezis, Konstantinos Tatas, Dimitrios Soudris, and Adonios Thanailakis. DAGGER: A novel generic methodology for FPGA bitstream generation and its software tool implementation. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 4–pp. IEEE, 2005. [cited on page 21]
- [260] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. [cited on page 160, 360]
- [261] Scott C. Smith, Ronald F. DeMara, Jiann S. Yuan, D. Ferguson, and D. Lamb. Optimization of null convention self-timed circuits. *INTEGRATION, the VLSI journal*, 37(3):135–165, 2004. [cited on page 253]
- [262] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985. [cited on page 47]
- [263] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 228–235. IEEE, 2011. [cited on page 21]
- [264] Danil Sokolov, Victor Khomenko, and Andrey Mokhov. Workcraft: ten years later. In *This Asynchronous World, Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 269–293. Newcastle University, 2016. [cited on page 18]

- [265] Ritesh K. Soni. *Open-Source Bitstream Generation for FPGAs*. PhD thesis, Virginia Tech, 2013. [cited on page 21]
- [266] Ritesh Kumar Soni, Neil Steiner, and Matthew French. Open-source bitstream generation. In *21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013. [cited on page 21]
- [267] J. Sparsø and S. Furber (eds.). *Principles of Asynchronous Circuit Design*. Springer, 2002. [cited on page 18, 32, 575]
- [268] Nattha Sretasereekul and Takashi Nanya. Eliminating isochronic-fork constraints in quasi-delay-insensitive circuits. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86(4):900–907, 2003. [cited on page 575]
- [269] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. Torc: towards an open-source tool flow. In *Proceedings of the 19th ACM/SIGDA international symposium on field programmable gate arrays*, pages 41–44. ACM, 2011. [cited on page 21]
- [270] Joseph E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT press, 1977. [cited on page 579]
- [271] Ted Sundstrom. Mathematical reasoning: Writing and proof. <https://www.tedsundstrom.com/mathreasoning>, 2018. [cited on page 88, 123]
- [272] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983. [cited on page 18]
- [273] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. [cited on page 27, 252, 485, 488]
- [274] Ivan E. Sutherland. The tyranny of the clock. *Communications of the ACM*, 55(10):35–36, October 2012. [cited on page 619]
- [275] Ivan E. Sutherland and Jo Ebergen. Computers without clocks. *Scientific American*, 287(2), August 2002. [cited on page 17]
- [276] Zoltán Gendler Szabó. Compositionality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2017. [cited on page 54]
- [277] Luca Tallini. Design of some new efficient balanced codes. Master’s thesis, Oregon State University, 1994. [cited on page 413]
- [278] Lev Vasilevich Tarasov. *The world is built on probability*. Mir, 1988. [cited on page 363]
- [279] Luis A. Tarazona, Doug A. Edwards, and Luis A. Plana. A synthesisable quasi-delay insensitive result forwarding unit for an asynchronous processor. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD’09. 12th Euromicro Conference on*, pages 627–634. IEEE, 2009. [cited on page 575]
- [280] R. Tarjan and M. Yannakakis. Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13:566–579, 1984. [cited on page 368]

- [281] Alexander Taubin, Jordi Cortadella, Luciano Lavagno, Alex Kondratyev, and Ad Peeters. Design automation of real-life asynchronous devices and systems. *Foundations and Trends in Electronic Design Automation*, 2(1):1–133, 2007. [cited on page 17, 575]
- [282] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Computers*, 24(5):418–428, 2007. [cited on page 17]
- [283] John Teifel and Rajit Manohar. An asynchronous dataflow FPGA architecture. *Computers, IEEE Transactions on*, 53(11):1376–1392, 2004. [cited on page 21]
- [284] Richard F. Tinder. Asynchronous sequential machine design and analysis: A comprehensive development of the design and analysis of clock-independent state machines and systems. *Synthesis Lectures on Digital Circuits and Systems*, 4(1):1–236, 2009. [cited on page 36]
- [285] William Benjamin Toms. *Synthesis of quasi-delay-insensitive datapath circuits*. PhD thesis, University of Manchester, 2006. [cited on page 253]
- [286] Stephen Trimberger. Effects of FPGA architecture on FPGA routing. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 574–578. ACM, 1995. [cited on page 21]
- [287] Alan Tucker. Polya’s enumeration formula by example. *Mathematics magazine*, 47(5):248–256, 1974. [cited on page 356]
- [288] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986. [cited on page 30, 35, 38, 47, 73, 137]
- [289] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *J. Exp. Algorithmics*, 15:1.6:1.1–1.6:1.64, February 2011. [cited on page 534]
- [290] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, pages 491–515. Springer, 1991. [cited on page 68]
- [291] Antti Valmari. State of the art report: Stubborn sets. *Petri Net Newsletter*, 46:6–14, 1994. [cited on page 68]
- [292] Antti Valmari and Henri Hansen. Can stubborn sets be optimal? *Fundamenta Informaticae*, 113(3):377–397, 2011. [cited on page 68]
- [293] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988. [cited on page 45, 185]
- [294] Tom Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994. [cited on page 19, 33, 47, 258, 298, 575]
- [295] Thomas Villiger. *Multi-point Interconnects for Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, ETH, Federal Institute of Technology Zurich, 2005. [cited on page 17, 258]
- [296] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008. [cited on page 55]

- [297] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook.org, 2013. [cited on page 55]
- [298] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968. [cited on page 222]
- [299] Jos H. Weber, Kees A. Schouhamer, Immink Hendrik, and C Ferreira. Extension of Knuth’s balancing algorithm with error correction. 2012. [cited on page 415]
- [300] Stephen Weston, Jean-Tristan Marin, James Spooner, Oliver Pell, and Oskar Mencer. Accelerating the computation of portfolios of tranching credit derivatives. In *Workshop on High Performance Computational Finance*, 2010. [cited on page 21]
- [301] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. [cited on page 587]
- [302] Skyler Windh, Xiaoyin Ma, Robert J. Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015. [cited on page 20]
- [303] Clifford Wolf and Mathias Lasser. Project IceStorm. <http://www.clifford.at/icestorm/>. [cited on page 20]
- [304] Catherine G. Wong, Alain J. Martin, and Peter Thomas. An architecture for asynchronous FPGAs. In *In Proceedings of International Conference on Field Programmable Technology*, pages 170–177, 2003. [cited on page 21]
- [305] Moe Thandar Wynn. *Semantics, verification, and implementation of workflows with cancellation regions and OR-joins*. PhD thesis, Queensland University of Technology, 2006. [cited on page 238]
- [306] Moe Thandar Wynn, H. M. W. Verbeek, Wil M. P van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Reduction rules for YAWL workflows with cancellation regions and OR-joins. *Information and Software Technology*, 51(6):1010–1020, 2009. [cited on page 238]
- [307] Liming Xiu. Clock technology: The next frontier. *IEEE Circuits and Systems Magazine*, 17(2):27–46, 2017. [cited on page 21]
- [308] Alex Yakovlev, Pascal Vivet, and Marc Renaudin. Advances in asynchronous logic: From principles to GALS & NoC, recent industry applications, and commercial CAD tools. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1715–1724. EDA Consortium, 2013. [cited on page 18]
- [309] Alexandre Yakovlev, Alexei Petrov, and Luciano Lavagno. A low latency arbitration circuit. *IEEE Transactions on VLSI Systems*, pages 372–377, 1994. [cited on page 258]
- [310] Alexandre Yakovlev, Alexei Petrov, and Luciano Lavagno. A low latency asynchronous arbitration circuit. *IEEE Transactions on VLSI Systems*, 2(3):372–377, September 1994. [cited on page 361]

- [311] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, February 1999. [cited on page 36]
- [312] Fabio Zanasi. *Interacting Hopf Algebras - the Theory of Linear Systems*. PhD thesis, Ecole Normale Supérieure de Lyon, 2015. [cited on page 205]