

Undergraduate Topics in Computer Science

Gerard O'Regan

Concise Guide to Formal Methods

Theory, Fundamentals and Industry
Applications



 Springer

The Springer logo consists of a stylized white chess knight piece on a black square, positioned to the left of the word "Springer" in a serif font.

Undergraduate Topics in Computer Science

Series editor

Ian Mackie

Advisory Boards

Samson Abramsky, University of Oxford, Oxford, UK

Karin Breitman, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

Chris Hankin, Imperial College London, London, UK

Dexter C. Kozen, Cornell University, Ithaca, USA

Andrew Pitts, University of Cambridge, Cambridge, UK

Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S. Skiena, Stony Brook University, Stony Brook, USA

Iain Stewart, University of Durham, Durham, UK

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at <http://www.springer.com/series/7592>

Gerard O'Regan

Concise Guide to Formal Methods

Theory, Fundamentals and Industry
Applications

Gerard O'Regan
SQC Consulting
Mallow, County Cork
Ireland

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-64020-4 ISBN 978-3-319-64021-1 (eBook)
DOI 10.1007/978-3-319-64021-1

Library of Congress Control Number: 2017946679

© Springer International Publishing AG 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*In memory of my dear aunt
Mrs. Noreen O'Regan*

Preface

Overview

The objective of this book is to give the reader a flavour of the formal methods field. The goal is to provide a broad and accessible guide to the fundamentals of formal methods, and to show how they may be applied to various areas in computing.

There are many existing books on formal methods, and while many of these provide more in-depth coverage on selected topics, this book is different in that it aims to provide a broad and accessible guide to the reader, as well as showing some of the rich applications of formal methods.

Each chapter of this book could potentially be a book in its own right, and so there are limits to the depth of coverage. However, the author hopes that this book will motivate and stimulate the reader, and encourage further study of the more advanced texts.

Organization and Features

Chapter 1 provides an introduction to the important field of software engineering. The birth of the discipline was at the Garmisch conference in Germany in the late 1960s. The extent to which mathematics should be employed in software engineering remains a topic of active debate.

Chapter 2 discusses software reliability and dependability, and covers topics such as software reliability and software reliability models; the Cleanroom methodology, system availability, safety and security critical systems, and dependability engineering.

Chapter 3 discusses formal methods, which consist of a set of mathematic techniques that provide an extra level of confidence in the correctness of the software. They may be employed to formally state the requirements of the proposed

system, and to derive a program from its mathematical specification. They allow a rigorous proof that the implemented program satisfies its specification to be provided, and they have been mainly applied to the safety critical field.

Chapter 4 provides an introduction to fundamental building blocks in discrete mathematics including sets, relations and functions. A set is a collection of well-defined objects, and it may be finite or infinite. A relation between two sets A and B indicates a relationship between members of the two sets, and is a subset of the Cartesian product of the two sets. A function is a special type of relation such that for each element in A there is at most one element in the co-domain B. Functions may be partial or total and injective, surjective or bijective.

Chapter 5 presents a short history of logic, and we discuss Greek contributions to syllogistic logic, stoic logic, fallacies and paradoxes. Boole's symbolic logic and its application to digital computing are discussed, and we consider Frege's work on predicate logic.

Chapter 6 provides an introduction to propositional and predicate logic. Propositional logic may be used to encode simple arguments that are expressed in natural language, and to determine their validity. The nature of mathematical proof is discussed, and we present proof by truth tables, semantic tableaux and natural deduction. Predicate logic allows complex facts about the world to be represented, and new facts may be determined via deductive reasoning. Predicate calculus includes predicates, variables and quantifiers, and a predicate is a characteristic or property that the subject of a statement can have.

Chapter 7 presents some advanced topics in logic including fuzzy logic, temporal logic, intuitionistic logic, undefined values, theorem provers and the applications of logic to AI. Fuzzy logic is an extension of classical logic that acts as a mathematical model for vagueness. Temporal logic is concerned with the expression of properties that have time dependencies, and it allows properties about the past, present and future to be expressed. Intuitionism was a controversial theory on the foundations of mathematics based on a rejection of the law of the excluded middle, and an insistence on constructive existence. We discuss three approaches to deal with undefined values, including the logic of partial functions; Dijkstra's approach with his *cand* and *cor* operators; and Parnas's approach which preserves a classical two-valued logic

Chapter 8 presents the Z specification language, which is one of the more popular formal methods. It was developed at the Programming Research Group at Oxford University in the early 1980s. Z specifications are mathematical, and the use of mathematics ensures precision, and allows inconsistencies and gaps in the specification to be identified. Theorem provers may be employed to demonstrate that the software implementation satisfies its specification.

Chapter 9 presents the Vienna Development Method, which is a popular formal specification language. We describe the history of its development at IBM in Vienna, and the main features of the language and its development method. Chapter 10 discusses the Irish school of VDM, which is a variant of classical VDM. We discuss its constructive mathematical approach, and where it differs from standard VDM.

Chapter 11 presents the unified modelling language (UML), which is a visual modelling language for software systems. It presents several views of the system architecture, and was developed at Rational Corporation as a notation for modelling object-oriented systems. We present various UML diagrams such as use case diagrams, sequence diagrams and activity diagrams.

Chapter 12 focuses on the approach of Dijkstra, Hoare and Parnas. We discuss the calculus of weakest preconditions developed by Dijkstra and the axiomatic semantics of programming languages developed by Hoare. We then discuss the classical engineering approach of Parnas, and his tabular expressions.

Chapter 13 discusses automata theory, including finite-state machines, push-down automata and Turing machines. Finite-state machines are abstract machines that are in only one state at a time, and the input symbol causes a transition from the current state to the next state. Pushdown automata have greater computational power than finite-state machines, and they contain extra memory in the form of a stack from which symbols may be pushed or popped. The Turing machine is the most powerful model for computation, and this theoretical machine is equivalent to an actual computer in the sense that it can compute exactly the same set of functions.

Chapter 14 discusses model checking which is an automated technique such that given a finite-state model of a system and a formal property, then it systematically checks whether the property is true or false in a given state in the model. It is an effective technique to identify potential design errors, and it increases the confidence in the correctness of the system design.

Chapter 15 discusses the nature of proof and theorem proving, and we discuss automated and interactive theorem provers. We discuss the nature of mathematical proof and formal mathematical proof.

Chapter 16 discusses probability and statistics and includes a discussion on discrete random variables; probability distributions; sample spaces; sampling; the abuse of statistics; variance and standard deviation; and hypothesis testing.

Chapter 17 discusses a selection of tools that are available to support the formal methodist in the performance of the various activities. Tools for VDM, Z, B, UML, theorem provers and model checking are considered.

Chapter 18 discusses technology transfer of formal methods to industry, and is concerned with the practical exploitation of new technology developed by an academic or industrial research group, and the objective is to facilitate its use of the technology in an industrial environment. Chapter 19 summarizes the journey that we have travelled in this book.

Audience

The audience of this book includes computer science students who wish to gain a broad and accessible overview of formal methods and its applications to the computing field. This book will also be of interest to students of mathematics who

are curious as to how formal methods are applied to the computing field. This book will also be of interest to the motivated general reader.

Acknowledgements

I am deeply indebted to family and friends who supported my efforts in this endeavour, and my thanks, as always, to the team at Springer. This book is dedicated to my late aunt (Mrs. Noreen O' Regan), who I always enjoyed visiting in Clonakilty, Co. Cork.

Cork, Ireland

Gerard O'Regan

Contents

1	Software Engineering	1
1.1	Introduction	1
1.2	What Is Software Engineering?	4
1.3	Challenges in Software Engineering	6
1.4	Software Processes and Life cycles	8
1.4.1	Waterfall Life cycle	9
1.4.2	Spiral Life cycles	10
1.4.3	Rational Unified Process	11
1.4.4	Agile Development	12
1.5	Activities in Waterfall Life cycle	14
1.5.1	Business Requirements Definition	15
1.5.2	Specification of System Requirements	16
1.5.3	Design	16
1.5.4	Implementation	17
1.5.5	Software Testing	17
1.5.6	Support and Maintenance	19
1.6	Software Inspections	20
1.7	Software Project Management	21
1.8	CMMI Maturity Model	21
1.9	Formal Methods	22
1.10	Review Questions	23
1.11	Summary	23
2	Software Reliability and Dependability	25
2.1	Introduction	25
2.2	Software Reliability	26
2.2.1	Software Reliability and Defects	27
2.2.2	Cleanroom Methodology	29
2.2.3	Software Reliability Models	30
2.3	Dependability	32
2.4	Computer Security	34
2.5	System Availability	35
2.6	Safety Critical Systems	36

2.7	Review Questions	37
2.8	Summary	37
3	Overview of Formal Methods	39
3.1	Introduction	39
3.2	Why Should We Use Formal Methods?	41
3.3	Industrial Applications of Formal Methods	42
3.4	Industrial Tools for Formal Methods	43
3.5	Approaches to Formal Methods	45
	3.5.1 Model-Oriented Approach	45
	3.5.2 Axiomatic Approach	46
3.6	Proof and Formal Methods	47
3.7	Mathematics in Software Engineering	48
3.8	The Vienna Development Method	48
3.9	VDM ⁺ , the Irish School of VDM	50
3.10	The Z Specification Language	51
3.11	The <i>B</i> -Method	52
3.12	Predicate Transformers and Weakest Preconditions	53
3.13	The Process Calculi	54
3.14	Finite-State Machines	55
3.15	The Parnas Way	55
3.16	Model Checking	56
3.17	Usability of Formal Methods	56
3.18	Review Questions	58
3.19	Summary	59
4	Sets, Relations and Functions	61
4.1	Introduction	61
4.2	Set Theory	62
	4.2.1 Set Theoretical Operations	64
	4.2.2 Properties of Set Theoretical Operations	67
	4.2.3 Russell's Paradox	68
	4.2.4 Computer Representation of Sets	69
4.3	Relations	70
	4.3.1 Reflexive, Symmetric and Transitive Relations	71
	4.3.2 Composition of Relations	73
	4.3.3 Binary Relations	75
	4.3.4 Applications of Relations	76
4.4	Functions	78
4.5	Application of Functions	82
	4.5.1 Miranda Functional Programming Language	84
4.6	Review Questions	85
4.7	Summary	86

5	A Short History of Logic	89
5.1	Introduction	89
5.2	Syllogistic Logic.	90
5.3	Paradoxes and Fallacies	91
5.4	Stoic Logic	93
5.5	Boole’s Symbolic Logic	95
5.5.1	Switching Circuits and Boolean Algebra	97
5.6	Application of Symbolic Logic to Digital Computing.	99
5.7	Frege	101
5.8	Review Questions	102
5.9	Summary	103
6	Propositional and Predicate Logic	105
6.1	Introduction	105
6.2	Propositional Logic.	106
6.2.1	Truth Tables	107
6.2.2	Properties of Propositional Calculus	109
6.2.3	Proof in Propositional Calculus	111
6.2.4	Semantic Tableaux in Propositional Logic.	114
6.2.5	Natural Deduction	116
6.2.6	Sketch of Formalization of Propositional Calculus	118
6.2.7	Applications of Propositional Calculus	118
6.2.8	Limitations of Propositional Calculus	120
6.3	Predicate Calculus	121
6.3.1	Sketch of Formalization of Predicate Calculus	123
6.3.2	Interpretation and Valuation Functions	125
6.3.3	Properties of Predicate Calculus	126
6.3.4	Applications of Predicate Calculus	127
6.3.5	Semantic Tableaux in Predicate Calculus	127
6.4	Review Questions	130
6.5	Summary	131
7	Advanced Topics in Logic	133
7.1	Introduction	133
7.2	Fuzzy Logic	134
7.3	Temporal Logic	135
7.4	Intuitionist Logic	137
7.5	Undefined Values	138
7.5.1	Logic of Partial Functions	139
7.5.2	Parnas Logic	141
7.5.3	Dijkstra and Undefinedness	142
7.6	Logic and AI	144
7.7	Theorem Provers for Logic.	147
7.8	Review Questions	149
7.9	Summary	150

8	Z Formal Specification Language	151
8.1	Introduction	151
8.2	Sets	153
8.3	Relations	154
8.4	Functions	156
8.5	Sequences	158
8.6	Bags	159
8.7	Schemas and Schema Composition	160
8.8	Reification and Decomposition	162
8.9	Proof in Z	164
8.10	Industrial Applications of Z	164
8.11	Review Questions	165
8.12	Summary	165
9	Vienna Development Method	167
9.1	Introduction	167
9.2	Sets	170
9.3	Sequences	171
9.4	Maps	173
9.5	Logic of Partial Functions in VDM	174
9.6	Data Types and Data Invariants	175
9.7	Specification in VDM	176
9.8	Refinement in VDM	177
9.9	Industrial Applications of VDM	178
9.10	Review Questions	178
9.11	Summary	179
10	Irish School of VDM	181
10.1	Introduction	181
10.2	Mathematical Structures and Their Morphisms	183
10.3	Models and Modelling	185
10.4	Sets	186
10.5	Relations and Functions	187
10.6	Sequences	189
10.7	Indexed Structures	191
10.8	Specifications and Proofs	191
10.9	Refinement in Irish VDM	193
10.10	Review Questions	196
10.11	Summary	196
11	Unified Modelling Language	199
11.1	Introduction	199
11.2	Overview of UML	200
11.3	UML Diagrams	202
11.4	Object Constraint Language	208

11.5	Industrial Tools for UML	209
11.6	Rational Unified Process	209
11.7	Review Questions	211
11.8	Summary	211
12	Dijkstra, Hoare and Parnas	213
12.1	Introduction	213
12.2	Calculus of Weakest Preconditions	218
12.2.1	Properties of WP	220
12.2.2	WP of Commands	221
12.2.3	Formal Program Development with WP	224
12.3	Axiomatic Definition of Programming Languages	225
12.4	Tabular Expressions	230
12.5	Review Questions	234
12.6	Summary	234
13	Automata Theory	235
13.1	Introduction	235
13.2	Finite-State Machines	236
13.3	Pushdown Automata	239
13.4	Turing Machines	241
13.5	Review Questions	243
13.6	Summary	243
14	Model Checking	245
14.1	Introduction	245
14.2	Modelling Concurrent Systems	248
14.3	Linear Temporal Logic	250
14.4	Computational Tree Logic	251
14.5	Tools for Model Checking	252
14.6	Industrial Applications of Model Checking	252
14.7	Review Questions	253
14.8	Summary	253
15	The Nature of Theorem Proving	255
15.1	Introduction	255
15.2	Early Automation of Proof	257
15.3	Interactive Theorem Provers	259
15.4	A Selection of Theorem Provers	261
15.5	Review Questions	262
15.6	Summary	262

16	Probability and Statistics	265
16.1	Introduction	265
16.2	Probability Theory	266
16.2.1	Laws of Probability	267
16.2.2	Random Variables	268
16.3	Statistics	271
16.3.1	Abuse of Statistics	271
16.3.2	Statistical Sampling	272
16.3.3	Averages in a Sample	273
16.3.4	Variance and Standard Deviation	274
16.3.5	Bell-Shaped (Normal) Distribution	274
16.3.6	Frequency Tables, Histograms and Pie Charts	277
16.3.7	Hypothesis Testing	278
16.4	Review Questions	280
16.5	Summary	280
17	Industrial Tools for Formal Methods	283
17.1	Introduction	283
17.2	Tools for Z	284
17.3	Tools for VDM	285
17.4	Tools for B	286
17.5	Tools for UML	287
17.6	Tools for Model Checking	288
17.7	Tools for Theorem Provers	288
17.8	Review Questions	289
17.9	Summary	290
18	Technology Transfer to Industry	291
18.1	Introduction	291
18.2	Formal Methods and Industry	292
18.3	Usability of Formal Methods	294
18.3.1	Why Are Formal Methods Difficult?	294
18.3.2	Characteristics of a Usable Formal Method	295
18.4	Pilot of Formal Methods	296
18.4.1	Technology Transfer of Formal Methods	296
18.5	Review Questions	298
18.6	Summary	298
19	Epilogue	299
19.1	The Future of Formal Methods	302
	References	303
	Index	309

Abbreviations

ACL2	A Computational Logic for Applicative Common Lisp
ACM	Association for Computing Machinery
AI	Artificial Intelligence
AMN	Abstract machine notation
APL	A programming language
ATM	Automated teller machine
ATP	Automated theorem prover
BCS	British Computer Society
CCS	Calculus communicating systems
BSL	Bandera Specification Language
CICS	Customer Information Control System
CMG	Computer Management Group
CMM	Capability Maturity Model
CMMI®	Capability Maturity Model Integration
COPQ	Cost of poor quality
COTS	Customized off the shelf
CSP	Communicating Sequential Processes
CTL	Computational tree logic
CZT	Community of Z Tools
DPDA	Deterministic pushdown automata
DSDM	Dynamic System Development Method
ESA	European Space Agency
FSM	Finite-state machine
GNU	GNU's Not Unix
GUI	Graphical user interface
HOL	Higher-order logic
IBM	International Business Machines
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IFIP	International Federation for Information Processing
ISEB	Information System Examination Board

ISO	International Standards Organization
ITP	Interactive theorem prover
JAD	Joint Application Development
KLOC	Thousand lines of code
LD	Limited domain
LEM	Law of the excluded middle
LISP	List processing
LPF	Logic of partial functions
LT	Logic Theorist
LTL	Linear temporal logic
MIT	Massachusetts Institute of Technology
MSL	Mars Science Laboratory
MTBF	Mean time between failure
MTTF	Mean time to failure
MOD	Ministry of Defence
NATO	North Atlantic Treaty Organization
NFA	Non-deterministic finite-state automaton
NQTHM	New Quantified THEorem prover
NRL	Naval Research Laboratory
OCL	Object constraint language
OMT	Object Modelling Technique
OTTER	Organized Techniques for Theorem-proving and Effective Research
PIN	Personal identity number
PL/1	Programming Language 1
PDA	Pushdown automata
PVS	Prototype Verification System
RAD	Rapid Application Development
RDBMS	Relational Database Management System
RSA	Rational Software Architect
RSM	Rational Software Modeler
RSRE	Royal Signals and Radar Establishment
RUP	Rational Unified Process
SAM	Semi-automated mathematics
SCAMPI	Standard CMMI Appraisal Method for Process Improvement
SDI	Strategic Defence Initiative
SEI	Software Engineering Institute
SPICE	Software Process Improvement Capability dEtermination
SQL	Structured Query Language
SRI	Stanford Research Institute
TDD	Test-driven development
TPS	Theorem Proving System
UAT	User acceptance testing
UML	Unified modelling language

VDM	Vienna Development Method
VDM ⁺	Irish School of VDM
VDM-SL	VDM specification language
WFF	Well-formed formula
Y2K	Year 2000

1.1 Introduction

The approach to software development in the 1950s and 1960s has been described as the “*Mongolian Hordes Approach*” by Fred Brooks [Brk:75].¹ The “method” or lack of method was characterized by the use of a large number of inexperienced programmers to fix a problem rather than solving it with a team of skilled programmers (i.e. throwing people at a problem). The view of software development at that time was characterized by:

- The completed code will always be full of defects.
- The coding should be finished quickly to correct these defects.
- Design as you code approach.

This philosophy accepted defeat in software development and suggested that irrespective of a solid engineering approach, that the completed software would always contain lots of defects, and that it therefore made sense to code as quickly as possible, and then to identify the defects that were present, so as to correct them as quickly as possible to solve a problem.

In the late 1960s, it was clear that the existing approaches to software development were deeply flawed, and that there was an urgent need for change. The NATO Science Committee organized two famous conferences to discuss critical issues in software development [Bux:75], with the first conference held at Garmisch, Germany, in 1968, and it was followed by a second conference in Rome in 1969.

Over fifty people from eleven countries attended the Garmisch conference, including Edsger Dijkstra, who did important theoretical work on formal specification and verification. The NATO conferences highlighted problems that existed in the software sector in the late 1960s, and the term “*software crisis*” was coined to

¹The “Mongolian Hordes” management myth is the belief that adding more programmers to a software project that is running late will allow it to catch-up. The reality is that adding people to a late software project actually makes it later.

refer to these. There were problems with budget and schedule overruns, as well as the quality and reliability of the delivered software.

The conference led to the birth of *software engineering* as a discipline in its own right, and the realization that programming is quite distinct from science and mathematics. Programmers are like engineers in that they build software products, and they therefore need education in traditional engineering as well as the latest technologies. The education of a classical engineer includes product design and mathematics. However, often computer science education has placed an emphasis on the latest technologies, rather than the important engineering foundations of designing and building high-quality products that are safe for the public to use.

Programmers therefore need to learn the key engineering skills to enable them to build products that are safe for the public to use. This includes a solid foundation on design, and on the mathematics required for building safe software products. Mathematics plays an important role in classical engineering, and it is also potentially useful in supporting software engineers in the delivery of high-quality software products in specialized domains such as safety critical systems. Several mathematical approaches to assist software engineers are described in [ORg:16b].

There are parallels between the software crisis of the late 1960s and the crisis with bridge construction in the nineteenth century. Several bridges collapsed, or were delivered late or over budget, due to the fact that people involved in their design and construction did not have the required engineering knowledge. This led to bridges that were poorly designed and constructed, and this led to their collapse and loss of life, as well as endangering the lives of the public.

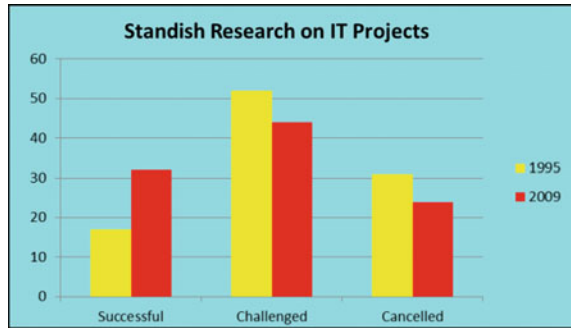
This led to legislation requiring engineers to be licensed by the Professional Engineering Association prior to practicing as engineers. This organization specified a core body of knowledge that the engineer is required to possess, and the licensing body verifies that the engineer has the required qualifications and experience. This helps to ensure that only personnel competent to design and build products actually do so. Engineers have a professional responsibility to ensure that the products are properly built and are safe for the public to use.

The Standish group has conducted research (Fig. 1.1) on the extent of problems with IT projects since the mid-1990s [Std:99]. These studies were conducted in the USA, but there is no reason to believe that European or Asian companies perform any better. The results indicate serious problems with on-time delivery of projects, and projects being cancelled prior to completion.² However, the comparison between 1995 and 2009 suggests that there have been some improvements with a greater percentage of projects being delivered successfully, and a reduction in the percentage of projects being cancelled.

Fred Brooks argues that software is inherently complex, and that there is no *silver bullet* that will resolve all of the problems associated with software development such as schedule or budget overruns [Brk:75, Brk:86]. Poor software

²These are IT projects covering diverse sectors including banking, telecommunications, etc., rather than pure software companies. Software companies following maturity frameworks such as the CMMI generally achieve more consistent results.

Fig. 1.1 Standish report—results of 1995 and 2009 survey



quality can lead to defects in the software that may adversely impact the customer, and even lead to loss of life. It is therefore essential that software development organizations place sufficient emphasis on quality throughout the software development life cycle.

The Y2K problem was caused by a two-digit representation of dates, and it required major rework to enable legacy software to function for the new millennium. Clearly, well-designed programs would have hidden the representation of the date, which would have required minimal changes for year 2000 compliance. Instead, companies spent vast sums of money to rectify the problem.

The quality of software produced by some companies is impressive.³ These companies employ mature software processes and are committed to continuous improvement. There is a lot of industrial interest in software process maturity models for software organizations, and various approaches to assess and mature software companies are described in [ORg:10, ORg:14].⁴ These models focus on improving the effectiveness of the management, engineering and organization practices related to software engineering, and in introducing best practice in software engineering. The disciplined use of the mature software processes by the software engineers enables high-quality software to be consistently produced.

The next section examines the nature of software engineering, and there is a more detailed account in the companion book “*Concise Guide to Software Engineering*” [ORg:17].

³I recall projects at Motorola that regularly achieved 5.6σ -quality in a L4 CMM environment (i.e. approx. 20 defects per million lines of code. This represents very high quality).

⁴Approaches such as the CMM or SPICE (ISO 15504) focus mainly on the management and organizational practices required in software engineering. The emphasis is on defining software processes that are fit for purpose and consistently following them. The process maturity models focus on what needs to be done rather how it should be done. This gives the organization the freedom to choose the appropriate implementation to meet its needs. The models provide useful information on practices to consider in the implementation.

1.2 What Is Software Engineering?

Software engineering involves the multi-person construction of multi-version programs. The IEEE 610.12 definition is:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of such approaches.

Software engineering includes:

1. Methodologies to design, develop and test software to meet customers' needs.
2. Software is engineered. That is, the software products are properly designed, developed and tested in accordance with engineering principles.
3. Quality and safety are properly addressed.
4. Mathematics may be employed to assist with the design and verification of software products. The level of mathematics employed will depend on the *safety critical* nature of the product. Systematic peer reviews and rigorous testing will often be sufficient to build quality into the software, with heavy *mathematical techniques reserved for safety and security critical software*.
5. Sound project management and quality management practices are employed.
6. Support and maintenance of the software is properly addressed.

Software engineering is not just programming. It requires the engineer to state precisely the requirements that the software product is to satisfy and then to produce designs that will meet these requirements. The project needs to be planned and delivered on time and budget. The requirements must provide a precise description of the problem to be solved; that is, *it should be evident from the requirements what is and what is not required*.

The requirements need to be rigorously reviewed to ensure that they are stated clearly and unambiguously and reflect the customer's needs. The next step is then to create the design that will solve the problem, and it is essential to validate the correctness of the design. Next, the software code to implement the design is written, and peer reviews and software testing are employed to verify and validate the correctness of the software.

The verification and validation of the design is rigorously performed for safety critical systems, and it is sometimes appropriate to employ mathematical techniques for these systems. However, it will usually be sufficient to employ peer reviews or software inspections as these methodologies provide a high degree of rigour. This may include approaches such as Fagan inspections [Fag:76], Gilb inspections [Glb:94], or Prince 2's approach to quality reviews [OGC:04].

The term "*engineer*" is a title that is awarded on merit in classical engineering. It is generally applied only to people who have attained the necessary education and competence to be called engineers, and who base their practice on sound engineering principles. The title places responsibilities on its holder to behave professionally and ethically. Often in computer science, the term "*software engineer*"

is employed loosely to refer to anyone who builds things, rather than to an individual with a core set of knowledge, experience and competence.

Several computer scientists (such as Parnas⁵) have argued that computer scientists should be educated as engineers to enable them to apply appropriate scientific principles to their work. They argue that computer scientists should receive a solid foundation in mathematics and design, to enable them to have the professional competence to perform as engineers in building high-quality products that are safe for the public to use. The use of mathematics is an integral part of the engineer's work in other engineering disciplines, and so the *software engineer* should be able to use mathematics to assist in the modelling or understanding of the behaviour or properties of the proposed software system.

Software engineers need education⁶ on specification, design, turning designs into programs, software inspections and testing. The education should enable the software engineer to produce well-structured programs that are fit for purpose.

Parnas has argued that software engineers have responsibilities as professional engineers.⁷ They are responsible for designing and implementing high-quality and reliable software that is safe to use. They are also accountable for their decisions and actions⁸ and have a responsibility to object to decisions that violate professional standards. Engineers are required to behave professionally and ethically with their

⁵Parnas has made important contributions to computer science. He advocates a solid engineering approach with the extensive use of classical mathematical techniques in software development. He also introduced information hiding in the 1970s, which is now a part of object-oriented design.

⁶Software companies that are following approaches such as the CMM or ISO 9001 consider the education and qualification of staff prior to assigning staff to performing specific tasks. The appropriate qualifications and experience for the specific role are considered prior to appointing a person to carry out the role. Many companies are committed to the education and continuous development of their staff, and on introducing best practice in software engineering into their organization.

⁷The ancient Babylonians used the concept of accountability, and they employed a code of laws (known as the Hammurabi Code) c. 1750 BC. It included a law that stated that if a house collapsed and killed the owner, then the builder of the house would be executed.

⁸However, it is unlikely that an individual programmer would be subject to litigation in the case of a flaw in a program causing damage or loss of life. A comprehensive disclaimer of responsibility for problems rather than a guarantee of quality accompany most software products. Software engineering is a team-based activity involving many engineers in various parts of the project, and it would be potentially difficult for an outside party to prove that the cause of a particular problem is due to the professional negligence of a particular software engineer, as there are many others involved in the process such as reviewers of documentation and code and the various test groups. Companies are more likely to be subject to litigation, as a company is legally responsible for the actions of their employees in the workplace, and a company is a wealthier entity than one of its employees. The legal aspects of licensing software may protect software companies from litigation. However, greater legal protection for the customer can be built into the contract between the supplier and the customer for bespoke-software development.

clients. The membership of the professional engineering body requires the member to adhere to the code of ethics⁹ of the profession. Engineers in other professions are licensed, and therefore, Parnas argues that a similar licensing approach be adopted for professional software engineers¹⁰ to provide confidence that they are competent for the particular assignment. Professional software engineers are required to follow best practice in software engineering and the defined software processes.¹¹

Many software companies invest heavily in training, as the education and knowledge of its staff are essential to delivering high-quality products and services. Employees receive professional training related to the roles that they are performing, such as project management, service management and software testing. The fact that the employees are professionally qualified increases confidence in the ability of the company to deliver high-quality products and services. A company that pays little attention to the competence and continuous development of its staff will obtain poor results and suffer a loss of reputation and market share.

1.3 Challenges in Software Engineering

The challenge in software engineering is to deliver high-quality software on time and on budget to customers. The research done by the Standish group was discussed earlier in this chapter, and the results of their 1998 research (Fig. 1.2) on project cost overruns in the USA indicated that 33% of projects are between 21 and 50% overestimate, 18% are between 51 and 100% overestimate and 11% of projects are between 101 and 200% overestimate.

The accurate estimation of project cost, effort and schedule is a challenge in software engineering. Therefore, project managers need to determine how good their estimation process actually is and to make appropriate improvements. The use of software metrics is an objective way to do this, and improvements in estimation will be evident from a reduced variance between estimated effort and actual effort. The project manager will determine and report the actual effort versus estimated effort and schedule for the project.

⁹Many software companies have a defined code of ethics that employees are expected to adhere. Larger companies will wish to project a good corporate image and to be respected worldwide.

¹⁰The British Computer Society (BCS) has introduced a qualification system for computer science professionals that it used to show that professionals are properly qualified. The most important of these is the BCS Information Systems Examination Board (ISEB) which allows IT professionals to be qualified in service management, project management, software testing, and so on.

¹¹Software companies that are following the CMMI or ISO 9001 standards will employ audits to verify that the processes and procedures have been followed. Auditors report their findings to management and the findings are addressed appropriately by the project team and affected individuals.

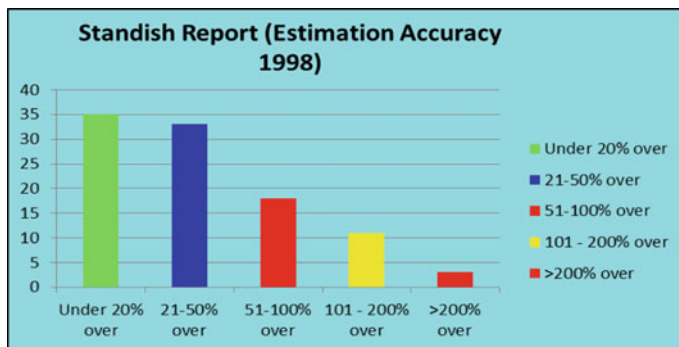


Fig. 1.2 Standish 1998 report—estimation accuracy

Risk management is an important part of project management, and the objective is to identify potential risks early and throughout the project and to manage them appropriately. The probability of each risk occurring and its impact is determined, and the risks are managed during project execution.

Software quality needs to be properly planned to enable the project to deliver a quality product. Flaws with poor-quality software lead to a negative perception of the company and could potentially lead to damage to the customer relationship with a subsequent loss of market share.

There is a strong economic case to building quality into the software, as less time is spent in reworking defective software. The cost of poor quality (COPQ) should be measured, and targets were set for its reductions. It is important that lessons are learned during the project and acted upon appropriately. This helps to promote a culture of continuous improvement.

A number of high-profile software failures are discussed in [ORg:14]. These include the millennium bug (Y2K) problem; the floating point bug in the Intel microprocessor; the European Space Agency Ariane-5 disaster. These failures led to embarrassment for the organizations involved, as well as the associated cost of replacement and correction.

The millennium bug was due to the use of two digits to represent dates rather than four digits. Its solution involved finding and analysing all code that had a Y2K impact; planning and making the necessary changes; and verifying the correctness of the changes. The worldwide cost of correcting the millennium bug is estimated to have been in billions of dollars.

The Intel Corporation was slow to acknowledge the floating point problem in its Pentium microprocessor, and in providing adequate information on the impact to its customers. It incurred a large financial cost in replacing microprocessors for its customers, as well as reputation damage. The Ariane-5 failure caused major embarrassment and damage to the credibility of the European Space Agency (ESA). Its maiden flight ended in failure on 4 June 1996, after a flight time of just 40 s.

These failures indicate that quality needs to be carefully considered when designing and developing software. The effect of software failure may be large costs in correcting and retesting the software, damage to the credibility and reputation of the company, or even loss of life.

1.4 Software Processes and Life cycles

Organizations vary by size and complexity, and the processes employed will reflect the nature of their business. The development of software involves many processes such as those for defining requirements; processes for project estimation and planning; processes for design, implementation, testing.

It is important that the processes employed are fit for purpose, and a key premise in the software quality field is that the quality of the resulting software is influenced by the quality and maturity of the underlying processes, and compliance to them. Therefore, it is necessary to focus on the quality of the processes as well as the quality of the resulting software.

There is, of course, little point in having high-quality processes unless their use is institutionalized in the organization. That is, all employees need to follow the processes consistently. This requires that the affected employees are trained on the processes, and that process discipline is instilled by an appropriate audit strategy that ensures compliance to them. Data will be collected to improve the process. The software process assets in an organization generally consist of:

- A software development policy for the organization
- Process maps that describe the flow of activities
- Procedures and guidelines that describe the processes in more detail
- Checklists to assist with the performance of the process
- Templates for the performance of specific activities (e.g. design, testing)
- Training Materials

The processes employed to develop high-quality software generally include:

- Project management process
- Requirements process
- Design process
- Coding process
- Peer-review process
- Testing process
- Supplier selection and management processes
- Configuration management process
- Audit process
- Measurement process
- Improvement process

- Customer support and maintenance processes

The software development process has an associated life cycle that consists of various phases. There are several well-known life cycles employed such as the waterfall model [Roy:70], the spiral model [Boe:88], the Rational Unified Process [Jac:99b] and the Agile methodology [Bec:00] which has become popular in recent years. The choice of a particular software development life cycle is determined from the particular needs of the specific project. The various life cycles are described in more detail in the following sections.

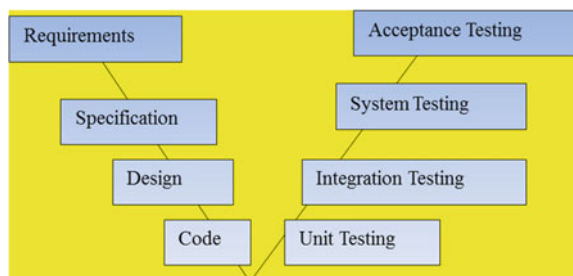
1.4.1 Waterfall Life cycle

The waterfall model (Fig. 1.3) starts with requirements gathering and definition. It is followed by the system specification (with the functional and non-functional requirements), the design and implementation of the software, and comprehensive testing. The software testing generally includes unit, system and user acceptance testing.

The waterfall model is employed for projects where the requirements can be identified early in the project life cycle or are known in advance. We are treating the waterfall model as identical to the “V” life cycle model, with the left-hand side of the “V” detailing requirements, specification, design and coding and the right-hand side detailing unit tests, integration tests, system tests and acceptance testing. Each phase has entry and exit criteria that must be satisfied before the next phase commences. There are several variations to the waterfall model.

Many companies employ a set of templates to enable the activities in the various phases to be consistently performed. Templates may be employed for project planning and reporting; requirements definition; design; testing and so on. These templates may be based on the IEEE standards or on industrial best practice.

Fig. 1.3 Waterfall V lifecycle model



1.4.2 Spiral Life cycles

The spiral model (Fig. 1.4) was developed by Barry Boehm in the 1980s [Boe:88], and it is useful for projects where the requirements are not fully known at project initiation, or where the requirements evolve as a part of the development life cycle. The development proceeds in a number of spirals, where each spiral typically involves objectives and an analysis of the risks, updates to the requirements, design, code, testing and a user review of the particular iteration or spiral.

The spiral is, in effect, a reusable prototype with the business analysts and the customer reviewing the current iteration and providing feedback to the development team. The feedback is analysed and used to plan the next iteration. This approach is often used in Joint Application Development, where the usability and look and feel of the application is a key concern. This is important in Web-based development and in the development of a graphical user interface (GUI). The implementation of part of the system helps in gaining a better understanding of the requirements of the system, and this feeds into subsequent development cycles. The process repeats until the requirements and the software product are fully complete.

There are several variations of the spiral model including Rapid Application Development (RAD); Joint Application Development (JAD) models; and the Dynamic Systems Development Method (DSDM) model. The Agile methodology (discussed in Sect. 1.4.4) has become popular in recent years, and it employs sprints

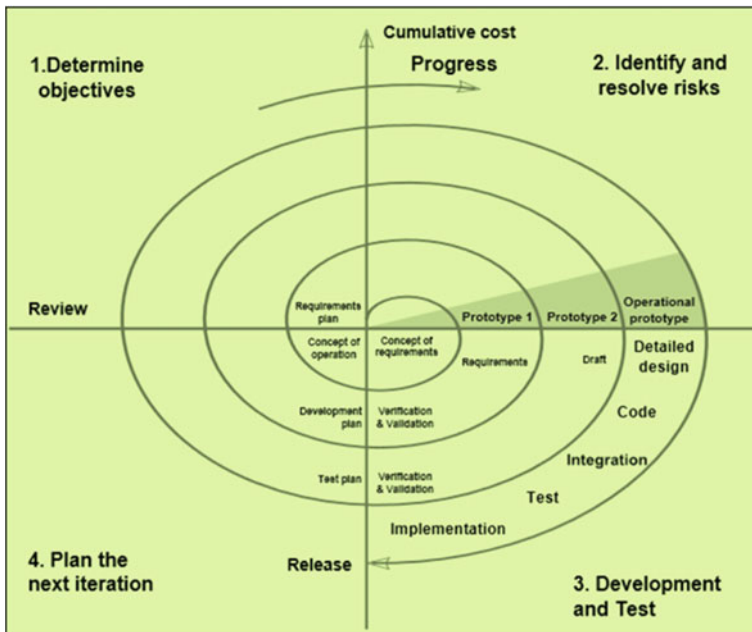


Fig. 1.4 Spiral lifecycle model ... public domain

(or iterations) of two weeks duration to implement a number of user stories. A sample spiral model is shown in Fig. 1.4.

There are other lifecycle models such as the iterative development process that combines the waterfall and spiral lifecycle model. The Cleanroom approach is discussed in Chap. 2, and it was developed by Harlan Mills at IBM. It includes a phase for formal specification, and its approach to software testing is based on the predicted usage of the software product, which enables a software reliability measure to be calculated. The Rational Unified Process (RUP) was developed by Rational, and it is discussed in the next section.

1.4.3 Rational Unified Process

The *Rational Unified Process* (RUP) was developed at the Rational Corporation (now part of IBM) in the late 1990s [Jac:99b]. It uses the unified modelling language (UML) as a tool for specification and design, where UML is a visual modelling language for software systems that provides a means of specifying, constructing and documenting the object-oriented system. RUP was developed by James Rumbaugh, Grady Booch and Ivar Jacobson, and it facilitates the understanding of the architecture and complexity of the system.

RUP is *use-case driven*, *architecture centric*, *iterative* and *incremental* and includes cycles, phases, workflows, risk mitigation, quality control, project management and configuration control (Fig. 1.5). Software projects may be very complex, and there are risks that requirements may be incomplete, or that the interpretation of a requirement may differ between the customer and the project team. RUP is a way to reduce risk in software engineering.

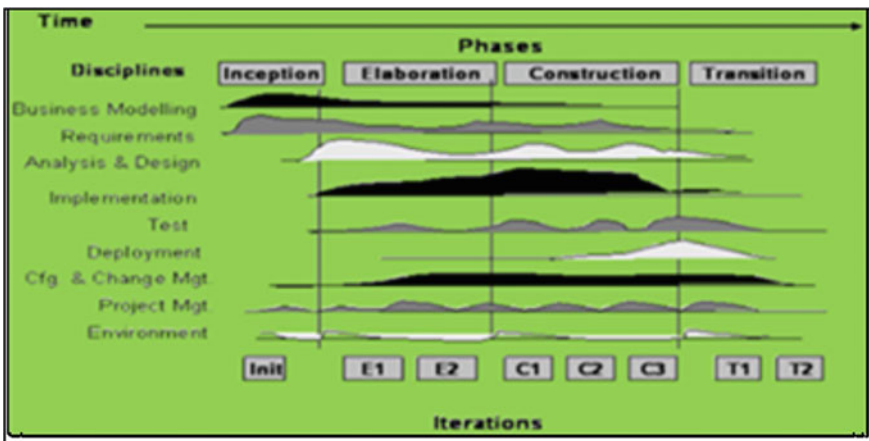


Fig. 1.5 Rational Unified Process

Requirements are gathered as use cases, where the *use cases describe the functional requirements from the point of view of the user of the system*. They describe what the system will do at a high level and ensure that there is an appropriate focus on the user when defining the scope of the project. *Use cases also drive the development process*, as the developers create a series of design and implementation models that realize the use cases. The developers review each successive model for conformance to the use-case model, and the test team verifies that the implementation correctly implements the use cases.

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the use cases and factors such as the platform that the software is to run on deployment considerations, legacy systems and the non-functional requirements.

RUP decomposes the work of a large project into smaller slices or mini-projects, and *each mini-project is an iteration that results in an increment to the product*. The iteration consists of one or more steps in the workflow and generally leads to the growth of the product. If there is a need to repeat an iteration, then all that is lost is the misdirected effort of one iteration, rather than the entire product. In other words, RUP is a way to mitigate risk in software engineering.

1.4.4 Agile Development

There has been a massive growth of popularity among software developers in lightweight methodologies like *Agile*. This is a software development methodology that is more responsive to customer needs than traditional methods such as the waterfall model. *The waterfall development model is similar to a wide and slow-moving value stream*, and halfway through the project, 100% of the requirements are typically 50% done. *However, for agile development, 50% of requirements are typically 100% done halfway through the project*.

This methodology has a strong collaborative style of working, and its approach includes:

- Aims to achieve a narrow fast-flowing value stream
- Feedback and adaptation employed in decision-making
- User stories and sprints are employed
- Stories are either done or not done (no such thing as 50% done)
- Iterative and incremental development is employed
- A project is divided into iterations
- An iteration has a fixed length (i.e. time boxing is employed)
- Entire software development life cycle is employed for the implementation of each story
- Change is accepted as a normal part of life in the Agile world
- Delivery is made as early as possible.
- Maintenance is seen as part of the development process
- Refactoring and evolutionary design Employed

- Continuous integration is employed
- Short cycle times
- Emphasis on quality
- Stand-up meetings
- Plan regularly
- Direct interaction preferred over documentation
- Rapid conversion of requirements into working functionality
- Demonstrate value early
- Early decision-making

Ongoing changes to requirements are considered normal in the Agile world, and it is believed to be more realistic to change requirements regularly throughout the project rather than attempting to define all of the requirements at the start of the project. The methodology includes controls to manage changes to the requirements, and good communication and early regular feedback is an essential part of the process.

A story may be a new feature or a modification to an existing feature. It is reduced to the minimum scope that can deliver business value, and a feature may give rise to several stories. Stories often build upon other stories, and the entire software development life cycle is employed for the implementation of each story. Stories are either done or not done; that is, there is such thing as a story being 80% done. The story is complete only when it passes its acceptance tests. Stories are prioritized based on a number of factors including as follows:

- Business value of Story
- Mitigation of risk
- Dependencies on other stories.

The Scrum approach is an Agile method for managing iterative development, and it consists of an outline planning phase for the project followed by a set of sprint cycles (where each cycle develops an increment). *Sprint planning* is performed before the start of the iteration, and stories are assigned to the iteration to fill the available time. Each scrum sprint is of a fixed length (usually 2–4 weeks), and it develops an increment of the system. The estimates for each story and their priority are determined, and the prioritized stories are assigned to the iteration. *A short morning stand-up meeting is held daily during the iteration and attended by the scrum master, the project manager¹² and the project team. It discusses the progress made the previous day, problem reporting and tracking, and the work planned for the day ahead. A separate meeting is held for issues that require more detailed discussion.*

Once the iteration is complete, the latest product increment is demonstrated to an audience including the product owner. This is to receive feedback and to identify new requirements. The team also conducts a retrospective meeting to identify what

¹²Agile teams are self-organizing, and the project manager role is generally not employed for small projects (<20 staff).

went well and what went poorly during the iteration. This is to for continuous improvement for future iterations. Planning for the next sprint then commences. The scrum master is a facilitator who arranges the daily meetings and ensures that the scrum process is followed. The role involves removing roadblocks so that the team can achieve their goals, and communicating with other stakeholders.

Agile employs pair programming and a collaborative style of working with the philosophy that two heads are better than one. This allows multiple perspectives in decision-making and a broader understanding of the issues.

Software testing is very important, and Agile generally employs automated testing for unit, acceptance, performance and integration testing. Tests are run frequently with the goal of catching programming errors early. They are generally run on a separate build server to ensure that all dependencies are checked. Tests are rerun before making a release. *Agile employs test-driven development with tests written before the code.* The developers write code to make a test pass with ideally developers only coding against failing tests. This approach forces the developer to write testable code.

Refactoring is employed in Agile as a design and coding practice. The objective is to change how the software is written without changing what it does. Refactoring is a tool for evolutionary design where the design is regularly evaluated, and improvements are implemented as they are identified. It helps in improving the maintainability and readability of the code and in reducing complexity. The automated test suite is essential in showing that the integrity of the software is maintained following refactoring.

Continuous integration allows the system to be built with every change. Early and regular integration allows early feedback to be provided. It also allows all of the automated tests to be run thereby identifying problems earlier. Agile is discussed in more detail in Chap. 18 of [ORg:17].

1.5 Activities in Waterfall Life cycle

The waterfall software development life cycle consists of various activities including as follows:

- Business (User) requirements definition
- Specification of system requirements
- Design
- Implementation
- Unit testing
- System testing
- UAT testing
- Support and maintenance

These activities are discussed in the following sections, and the description is specific to the non-Agile world.

1.5.1 Business Requirements Definition

The business (user) requirements specify what the customer wants and define what the software system is required to do (*as distinct from how this is to be done*). The requirements are the foundation for the system, and if they are incorrect, then the implemented system will be incorrect. *Prototyping may be employed* to assist in the definition and validation of the requirements. The process of determining the requirements, analysing and validating them and managing them throughout the project life cycle is termed *requirements engineering*.

The *user requirements* are determined from discussions with the customer to determine their actual needs, and they are then refined into the *system requirements*, which state the *functional* and *non-functional* requirements of the system. The specification of the user requirements needs to be unambiguous to ensure that all parties involved in the development of the system share a common understanding of what is to be developed and tested.

Requirements gathering involve meetings with the stakeholders to gather all relevant information for the proposed product. The stakeholders are interviewed, and requirements workshops conducted to elicit the requirements from them. An early working system (prototype) is often used to identify gaps and misunderstandings between developers and users. The prototype may serve as a basis for writing the specification.

The requirements workshops are used to discuss and prioritize the requirements, as well as identifying and resolving any conflicts between them. The collected information is consolidated into a coherent set of requirements. Changes to the requirements may occur during the project, and these need to be controlled. It is essential to understand the impacts (e.g. schedule, budget and technical) of a proposed change to the requirements prior to its approval.

Requirements verification is concerned with ensuring that the requirements are properly implemented (i.e. building it right) in the design and implementation. *Requirements validation* is concerned with ensuring that the right requirements are defined (building the right system), and that they are precise, complete and reflect the actual needs of the customer.

The requirements are validated by the stakeholders to ensure that they are actually those desired and to establish their feasibility. This may involve several reviews of the requirements until all stakeholders are ready to approve the requirements document. Other validation activities include reviews of the prototype and the design, and user acceptance testing.

The requirements for a system are generally documented in a natural language such as “English”. Other notations that are employed include the visual modelling language UML [Jac:99a] and formal specification languages such as VDM or Z for the safety critical field.

The Agile software development methodology was discussed earlier, and it argues that as requirements change so quickly that a requirements document is unnecessary, since such a document would be out of date as soon as it was written.

1.5.2 Specification of System Requirements

The specification of the system requirements of the product is essentially a statement of what the software development organization will provide to meet the business (user) requirements. That is, the detailed business requirements are a statement of what the customer wants, whereas the specification of the system requirements is a statement of what will be delivered by the software development organization.

It is essential that the system requirements are valid with respect to the user requirements, and they are reviewed by the stakeholders to ensure their validity. Traceability may be employed to show that the business requirements are addressed by the system requirements.

There are two categories of system requirements: namely functional and non-functional requirements. The *functional requirements* define the functionality that is required of the system, and it may include screenshots, report layouts or desired functionality specified as use cases. The *non-functional requirements* will generally include security, reliability, availability, performance and portability requirements, as well as usability and maintainability requirements.

1.5.3 Design

The design of the system consists of engineering activities to describe the architecture or structure of the system, as well as activities to describe the algorithms and functions required to implement the system requirements. It is a creative process concerned with how the system will be implemented, and its activities include architecture design, interface design and data structure design. There are often several possible design solutions for a particular system, and the designer will need to decide on the most appropriate solution.

The design may be specified in various ways such as graphical notations that display the relationships between the components making up the design. The notation may include flowcharts or various UML diagrams such as sequence diagrams, state charts. Program description languages or pseudo code may be employed to define the algorithms and data structures that are the basis for implementation.

Function-oriented design is mainly historical, and it involves starting with a high-level view of the system and refining it into a more detailed design. The system state is centralized and shared between the functions operating on that state.

Object-oriented design has become popular, and it is based on the concept of *information hiding* developed by Parnas [Par:72]. The system is viewed as a collection of objects rather than functions, with each object managing its own state information. The system state is decentralized, and an object is a member of a class. The definition of a class includes attributes and operations on class members, and these may be inherited from superclasses. Objects communicate by exchanging messages.

It is essential to verify and validate the design with respect to the system requirements, and this will be done by traceability of the design to the system requirements and design reviews.

1.5.4 Implementation

This phase is concerned with implementing the design in the target language and environment (e.g. C++ or Java), and it involves writing or generating the actual code. The development team divides up the work to be done, with each programmer responsible for one or more modules. The coding activities often include code reviews or walk-throughs to ensure that quality code is produced, and to verify its correctness. The code reviews will verify that the source code conforms to the coding standards and that maintainability issues are addressed. They will also verify that the code produced is a valid implementation of the software design.

Software reuse provides a way to speed up the development process. Components or objects that may be reused need to be identified and handled accordingly. The implemented code may use software components that have either being developed internally or purchased off the shelf. Open-source software has become popular in recent years, and it allows software developed by others to be used (*under an open-source licence*) in the development of applications.

The benefits of software reuse include increased productivity and a faster time to market. There are inherent risks with customized-off-the shelf (COTS) software, as the supplier may decide to no longer support the software, or there is no guarantee that software that has worked successfully in one domain will work correctly in a different domain. It is therefore important to consider the risks as well as the benefits of software reuse and open-source software.

1.5.5 Software Testing

Software testing is employed to verify that the requirements have been correctly implemented, and that the software is fit for purpose, as well as identifying defects present in the software. There are various types of testing that may be conducted including *unit testing*, *integration testing*, *system testing*, *performance testing* and *user acceptance testing*. These are described below:

Unit Testing

Unit testing is performed by the programmer on the completed unit (or module), and prior to its integration with other modules. These tests are written by the programmer, and the objective is to show that the code satisfies the design. The unit test case is generally documented, and it should include the test objective, the test procedure and the expected result.

Code coverage and branch coverage metrics are often generated to give an indication of how comprehensive the unit testing has been. These metrics provide visibility into the number of lines of code executed as well as the branches covered during unit testing.

The developer executes the unit tests, records the results, corrects any identified defects and retests the software. *Test-driven development* (TDD) has become popular (e.g. in the Agile world), and this involves writing the unit test case (and possibly other test cases) before the code, and the code is written to pass the defined test cases.

Integration Test

The development team performs this type of testing on the integrated system, once all of the individual units work correctly in isolation. The objective is to verify that all of the modules and their interfaces work correctly together, and to identify and resolve any issues. Modules that work correctly in isolation may fail when integrated with other modules. This type of testing is generally performed by the developers.

System Test

The purpose of system testing is to verify that the implementation is valid with respect to the system requirements. It involves the specification of system test cases, and their execution will verify that the system requirements have been correctly implemented. An independent test group generally conducts this type of testing, and the system tests are traceable to the system requirements.

Any system requirements that have been incorrectly implemented will be identified, and defects logged and reported to the developers. The test group will verify that the new version of the software is correct, and regression testing is conducted to verify system integrity. System testing may include security testing, usability testing and performance testing.

The preparation of the test environment requires detailed planning, and it may involve ordering special hardware and tools. It is important that the test environment is set up early to allow the timely execution of the test cases.

Performance Test

The purpose of performance testing is to ensure that the performance of the system is within the bounds specified by the non-functional requirements. It may include *load performance testing*, where the system is subjected to heavy loads over a long period of time, and *stress testing*, where the system is subjected to heavy loads during a short time interval.

Performance testing often involves the simulation of many users using the system and involves measuring the response times for various activities. Test tools are employed to simulate a large number of users and heavy loads. It is also employed to determine whether the system is scalable to support future growth.

User Acceptance Test

UAT testing is usually performed under controlled conditions at the customer site, and its operation will closely resemble the real-life behaviour of the system. The customer will see the product in operation and will judge whether or not the system is fit for purpose.

The objective is to demonstrate that the product satisfies the business requirements and meets the customer expectations. Upon its successful completion, the customer is happy to accept the product.

1.5.6 Support and Maintenance

This phase continues after the release of the software product to the customer. Software systems often have a long lifetime (e.g. consider the millions of lines of legacy COBOL programs), and the software needs to be continuously enhanced over its lifetime to meet the evolving needs of the customers. This may involve regular new releases with new functionality and corrections to known defects.

Any problems that the customer notes with the software are reported as per the customer support and maintenance agreement. The support issues will require investigation, and the issue may be *a defect in the software*, *an enhancement to the software*, or *due to a misunderstanding*. The support and maintenance team will identify the causes of any identified defects and will implement an appropriate solution to resolve. Testing is conducted to verify that the solution is correct, and that the changes made have not adversely affected other parts of the system. Mature organizations will conduct post-mortems to learn lessons from the defect¹³ and will take corrective action to prevent a reoccurrence.

The presence of a maintenance phase suggests an acceptance of the reality that problems with the software will be identified postrelease. The goal of building a correct and reliable software product the first time is very difficult to achieve, and the customer is always likely to find some issues with the released software product. It is accepted today that quality needs to be built into each step in the development process, with the role of software inspections and testing to identify as many defects as possible prior to release, and minimize the risk that that serious defects will be found postrelease.

The more effective the in-phase inspections of deliverables, the higher the quality of the resulting software, with a corresponding reduction in the number of defects. The testing group plays a key role in verifying that the system is correct, and in providing confidence that the software is fit for purpose and ready to be released. The approach to software correctness involves testing and retesting, until

¹³This is essential for serious defects that have caused significant inconvenience to customers (e.g. a major telecoms outage). The software development organization will wish to learn lessons to determine what went wrong in its processes that prevented the defect from being identified during peer reviews and testing. Actions to prevent a reoccurrence will be identified and implemented.

the testing group believes that all defects have been eliminated. Dijkstra [Dij:72] argued that:

Testing a program demonstrates that it contains errors, never that it is correct.

That is, irrespective of the amount of time spent on testing, it can never be said with absolute confidence that all defects have been found in the software. Testing provides increased confidence that the program is correct, and statistical techniques may be employed to give a measure of the software reliability.

Many software companies may consider one defect per thousand lines of code (KLOC) to be reasonable quality. However, if the system contains one million lines of code, this is equivalent to a thousand postrelease defects, which is unacceptable.

Some mature organizations have a quality objective of three defects per million lines of code, which was introduced by Motorola as part of its six-sigma (6σ) program. It was originally applied to its manufacturing businesses and subsequently applied to its software organizations. The goal is to reduce variability in manufacturing processes and to ensure that the processes performed within strict process control limits.

1.6 Software Inspections

Software inspections are used to build quality into software products. There are a number of well-known approaches such as the Fagan methodology [Fag:76], Gilb's approach [Gib:94] and Prince 2's approach.

Fagan inspections were developed by Michael Fagan of IBM. It is a seven-step process that identifies and removes errors in work products. The process mandates that requirement documents, design documents, source code and test plans are all formally inspected by experts independent of the author of the deliverable to ensure quality.

There are various *roles* defined in the process including the *moderator* who chairs the inspection. The *reader's* responsibility is to read or paraphrase the particular deliverable, and *the author* is the creator of the deliverable and has a special interest in ensuring that it is correct. The *tester* role is concerned with the test viewpoint.

The inspection process will consider whether the design is correct with respect to the requirements and whether the source code is correct with respect to the design. Software inspections play an important role in building quality into software and in reducing the cost of poor quality in the organization.

1.7 Software Project Management

The timely delivery of quality software requires good management and engineering processes. Software projects have a history of being delivered late or over budget, and good project management practices include the following activities:

- Estimation of cost, effort and schedule for the project
- Identifying and managing risks
- Preparing the project plan
- Preparing the initial project schedule and key milestones
- Obtaining approval for the project plan and schedule
- Staffing the project
- Monitoring progress, budget, schedule, effort, risks, issues, change requests and quality
- Taking corrective action
- Re-planning and rescheduling
- Communicating progress to affected stakeholders
- Preparing status reports and presentations

The project plan will contain or reference several other plans such as the project quality plan, the communication plan, the configuration management plan and the test plan.

Project estimation and scheduling are difficult as often software projects are breaking new ground and differ from previous projects. That is, previous estimates may often not be a good basis for estimation for the current project. Often, unanticipated problems can arise for technically advanced projects, and the estimates may often be optimistic. Gantt charts are often employed for project scheduling, and these show the work breakdown for the project, as well as task dependencies and allocation of staff to the various tasks.

The effective management of risk during a project is essential to project success. Risks arise due to uncertainty, and the risk management cycle involves¹⁴ risk identification; risk analysis and evaluation; identifying responses to risks; selecting and planning a response to the risk; and risk monitoring. The risks are logged, and the likelihood of each risk arising and its impact is then determined. Each risk is assigned an owner and an appropriate response determined.

1.8 CMMI Maturity Model

The CMMI is a framework to assist an organization in the implementation of best practice in software and systems engineering. It is an internationally recognized model for software process improvement and assessment and is used worldwide by

¹⁴These are the risk management activities in the Prince 2 methodology.

thousands of organizations. It provides a solid engineering approach to the development of software and supports the definition of high-quality processes for the various software engineering and management activities.

It was developed by the Software Engineering Institute (SEI) who adapted the process improvement principles used in the manufacturing field to the software field. They developed the original CMM model and its successor the CMMI. The CMMI states *what the organization needs to do* to mature its processes rather than *how this should be done*.

The CMMI consists of five maturity levels with each maturity level consisting of several process areas. Each process area consists of a set of goals, and these goals are implemented by practices related to that process area. Level two is focused on management practices; level three is focused on engineering and organization practices; level four is concerned with ensuring that key processes are performing within strict quantitative limits; level five is concerned with continuous process improvement. Maturity levels may not be skipped in the staged representation of the CMMI, as each maturity level is the foundation for the next level. The CMMI and Agile are compatible, and CMMI v1.3 supports Agile software development.

The CMMI allows organizations to benchmark themselves against other organizations. This is done by a formal SCAMPI appraisal conducted by an authorized lead appraiser. The results of the appraisal are generally reported back to the SEI, and there is a strict qualification process to become an *authorized lead appraiser*. An appraisal is useful in verifying that an organization has improved, and it enables the organization to prioritize improvements for the next improvement cycle. The CMMI is discussed in more detail in [CKS:11].

1.9 Formal Methods

Dijkstra and Hoare have argued that the way to develop correct software is to derive the program from its specifications using mathematics, and to employ *mathematical proof* to demonstrate its correctness with respect to the specification. This offers a rigorous framework to develop programs adhering to the highest quality constraints. However, in practice, mathematical techniques have proved to be cumbersome to use, and their widespread use in industry is unlikely at this time.

The *safety critical area* is one domain to which mathematical techniques have been successfully applied. There is a need for extra rigour in this field, and mathematical techniques can demonstrate the presence or absence of certain desirable or undesirable properties (e.g. “*when a train is in a level crossing, then the gate is closed*”).

Spivey [Spi:92] defines a “*formal specification*” as the use of mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved. It describes *what* the system must do, as distinct from *how* it is to be done. This abstraction away from implementation enables questions about what the

system does to be answered, independently of the detailed code. Further, the unambiguous nature of mathematical notation avoids the problem of ambiguity in an imprecisely worded natural language description of a system.

The formal specification thus becomes the key reference point for the different parties concerned with the construction of the system and is a useful way of promoting a common understanding for all those concerned with the system. The term “*formal methods*” is used to describe a formal specification language and a method for the design and implementation of computer systems.

The specification is written precisely in a mathematical language. The derivation of an implementation from the specification may be achieved via *stepwise refinement*. Each refinement step makes the specification more concrete and closer to the actual implementation. There is an associated *proof obligation* that the refinement be valid, and that the concrete state preserves the properties of the more abstract state. Thus, assuming the original specification is correct and the proofs of correctness of each refinement step are valid, then there is a very high degree of confidence in the correctness of the implemented software.

Formal methods have been applied to a diverse range of applications, including circuit design, Artificial Intelligence, specification of standards, specification and verification of programs. They are described in more detail in the remainder of the book.

1.10 Review Questions

1. Discuss the research results of the Standish group on IT project delivery?
2. What are the main challenges in software engineering?
3. Describe various software life cycles.
4. Discuss the advantages and disadvantages of Agile.
5. Describe the purpose of the CMMI? What are the benefits?
6. Describe the main activities in software inspections.
7. Describe the main activities in software testing.
8. Describe the main activities in project management?
9. What are the advantages and disadvantages of formal methods?

1.11 Summary

The birth of software engineering was at the NATO conference held in 1968 in Germany. This conference highlighted the problems that existed in the software sector in the late 1960s, and the term “*software crisis*” was coined to refer to these.

This led to the realization that programming is quite distinct from science and mathematics, and that software engineers need to be properly trained to enable them to build high-quality products that are safe to use.

The Standish group conducts research on the extent of problems with the delivery of projects on time and budget. Their research indicates that it remains a challenge to deliver projects on time, on budget and with the right quality.

Programmers are like engineers in the sense that they build products. Therefore, programmers need to receive an appropriate education in engineering as part of their training. The education of traditional engineers includes training on product design and an appropriate level of mathematics.

Software engineering involves multi-person construction of multi-version programs. It is a systematic approach to the development and maintenance of the software, and it requires a precise statement of the requirements of the software product, and then the design and development of a solution to meet these requirements. It includes methodologies to design, develop, implement and test software as well as sound project management, quality management and configuration management practices. Support and maintenance of the software needs to be properly addressed. There is a more detailed account of software engineering in “*Concise Guide to Software Engineering*” [ORg:17].

2.1 Introduction

This chapter gives an introduction to the important area of software reliability and dependability, and it discusses important topics in software engineering such as software reliability; software availability; software reliability models; the Cleanroom methodology; dependability and its various dimensions; security engineering; and safety critical systems.

Software reliability is the probability that the program works without failure for a period of time, and it is usually expressed as the mean time to failure. It is different from hardware reliability, in that hardware is characterized by components that physically wear out, whereas software is intangible and software failures are due to design and implementation errors. In other words, software is either correct or incorrect when it is designed and developed, and it does not physically deteriorate with time.

Harlan Mills and others at IBM developed the Cleanroom approach to software development, and the process is described in [ORg:06]. It involves the application of statistical techniques to calculate a software reliability measure based on the expected usage of the software.¹ This involves executing tests chosen from the population of all possible uses of the software in accordance with the probability of its expected use. Statistical usage testing is more effective in finding defects that lead to failure than coverage testing.

Models are simplifications of the reality, and a good model allows accurate predictions of future behaviour to be made. A model is judged effective if there is good empirical evidence to support it, and a good software reliability model will have good theoretical foundations and realistic assumptions. The extent to which the software reliability model can be trusted depends on the accuracy of its predictions, and empirical data will need to be gathered to judge its accuracy. A good

¹The expected usage of the software (or operational profile) is a quantitative characterization (usually based on probability) of how the system will be used.

software reliability model will give good predictions of the reliability of the software.

It is essential that software that is widely used is dependable, which means that the software is available whenever required, and that it operates safely and reliably without any adverse side effects. Today, billions of computers are connected to the Internet, and this has led to a growth in attacks on computers. It is essential that computer security is carefully considered, and developers need to be aware of the threats facing a system and techniques to eliminate them. The developers need to be able to develop secure systems that are able to deal with and recover from external attacks.

2.2 Software Reliability

The design and development of high-quality software has become increasingly important for society. The hardware field has been very successful in developing sound reliability models, which allow useful predictions of how long a hardware component (or product) will function to be provided. This has led to a growing interest in the software field in the development of a sound software reliability model. Such a model would provide a sound mechanism to predict the reliability of the software prior to its deployment at the customer site, as well as confidence that the software is fit for purpose and safe to use.

Definition 2.1 (*Software Reliability*)

Software reliability is the probability that the program works without failure for a specified length of time, and it is a statement of the future behaviour of the software. It is generally expressed in terms of the *mean time to failure* (MTTF) or the *mean time between failure* (MTBF).

Statistical sampling techniques are often employed to predict the reliability of hardware, as it is not feasible to test all items in a production environment. The quality of the sample is then used to make inferences on the quality of the entire population, and this approach is effective in manufacturing environments where variations in the manufacturing process often lead to defects in the physical products.

There are similarities and differences between hardware and software reliability. A hardware failure generally arises due to a component wearing out due to its age, and often a replacement component is required. Many hardware components are expected to last for a certain period of time, and the variation in the failure rate of a hardware component is often due to variations in the manufacturing process, and to the operating environment of the component. Good hardware reliability predictors have been developed, and each hardware component has an expected mean time to failure. The reliability of a product may then be determined from the reliability of the individual components.

Software is an intellectual undertaking involving a team of designers and programmers. It does not physically wear out as such, and software failures manifest themselves from particular user inputs. Each copy of the software code is identical, and the software code is either correct or incorrect. That is, software failures are due to design and implementation errors, rather than to the software physically wearing out over time. The software community has not yet developed a sound software reliability predictor model.

The software population to be sampled consists of all possible execution paths of the software, and since this is potentially infinite, it is generally not possible to perform exhaustive testing. The way in which the software is used (i.e. the inputs entered by the users) will impact upon its perceived reliability. Let I_f represent the fault set of inputs (i.e. $i_f \in I_f$ if and only if the input of i_f by the user leads to failure). The randomness of the time to software failure is due to the unpredictability in the selection of an input $i_f \in I_f$. It may be that the elements in I_f are inputs that are rarely used, and therefore, the software will be perceived as reliable.

Statistical usage testing may be used to make predictions on the future performance and reliability of the software. This requires an understanding of the expected usage profile of the system, as well as the population of all possible usages of the software. The sampling is done in accordance with the expected usage profile, and a software reliability measure is calculated.

2.2.1 Software Reliability and Defects

The release of an unreliable software product may result in damage to property or injury (including loss of life) to a third party. Consequently, companies need to be confident that their software products are fit for use prior to their release. The project team needs to conduct extensive inspections and testing of the software, as well as considering all associated risks prior to its release.

Objective product quality criteria may be set (e.g. 100% of tests performed and passed) that must be satisfied prior to the release of the product. This provides a degree of confidence that the software has the desired quality, and is fit for purpose. However, these results are historical in the sense that they are a statement of past and present quality. The question is whether the past behaviour and performance provides a sound indication of future behaviour.

Software reliability models are an attempt to predict the future reliability of the software, and to assist in deciding on whether the software is ready for release. A defect does not always result in a failure, as it may occur on a rarely used execution path. Studies indicate that many observed failures arise from a small proportion of the existing defects.

Adam's 1984 case study [Ada:84] indicates that over 33% of the defects led to an observed failure with mean time to failure greater than 5000 years, whereas less than 2% of defects led to an observed failure with a mean time to failure of less than 5 years. This suggests that a small proportion of defects often lead to almost all of the observed failures (Table 2.1).

Table 2.1 Adam's 1984 study of software failures of IBM products

Rare					Frequent			
	1	2	3	4	5	6	7	8
MTTF (years)	5000	1580	500	158	50	15.8	5	1.58
Avg % fixes	33.4	28.2	18.7	10.6	5.2	2.5	1.0	0.4
Prob failure	0.008	0.021	0.044	0.079	0.123	0.187	0.237	0.300

The analysis shows that 61.6% of all fixes (groups 1 and 2) were for failures that will be observed less than once in 1580 years of expected use, and that these constitute only 2.9% of the failures observed by typical users. On the other hand, groups 7 and 8 constitute 53.7% of the failures observed by typical users and only 1.4% of fixes.

This case study showed that *coverage testing* is not cost effective in increasing MTTF. *Usage testing*, in contrast, would allocate 53.7% of the test effort to fixes that will occur 53.7% of the time for a typical user. Harlan Mills has argued [CoM:90] that the data in the table shows that usage testing is 21 times more effective than coverage testing.

There is a need to be careful with *reliability growth models*, as there is no tangible growth in reliability unless the corrected defects are likely to manifest themselves as a failure.² Many existing software reliability growth models assume that all remaining defects in the software have an equal probability of failure, and that the correction of a defect leads to an increase in software reliability. These assumptions are questionable.

The defect count and defect density may be poor predictors of operational reliability, and an emphasis on removing a large number of defects from the software may not be sufficient to achieve high reliability.

The correction of defects in the software leads to a newer version of the software, and many software reliability models assume reliability growth; i.e. the new version is more reliable than the older version as several identified defects have been corrected. However, in some sectors such as the safety critical field, the view is that the new version of a program is a new entity, and that no inferences may be drawn until further investigation has been done. There are a number of ways to interpret the relationship between the new version of the software and the older version (Table 2.2).

The safety critical industry (e.g. the nuclear power industry) takes the conservative viewpoint that any change to a program creates a new program. The new program is therefore required to demonstrate its reliability, and so extensive testing needs to be performed.

²We are assuming that the defect has been corrected perfectly with no new defects introduced by the changes made.

Table 2.2 New and old version of software

 Similarities and differences between new/old version

- The new version of the software is identical to the previous version except that the identified defects have been corrected
 - The new version of the software is identical to the previous version, except that the identified defects have been corrected, but the developers have introduced some new defects
 - No assumptions can be made about the behaviour of the new version of the software until further data is obtained
-

2.2.2 Cleanroom Methodology

Harlan Mills and others at IBM developed the Cleanroom methodology as a way to develop high-quality software [CoM:90]. Cleanroom helps to ensure that the software is released only when it has achieved the desired quality level, and the probability of zero defects is very high.

The way in which the software is used will impact on its perceived quality and reliability. Failures will manifest themselves on certain input sequences, and as the input sequences will vary among users, the result will be different perceptions of the reliability of the software among the users. The knowledge of how the software will be used allows the software testing to focus on verifying the correctness of common everyday tasks carried out by users.

Therefore, it is important to determine the operational profile of the users to enable effective software testing to be performed. This may be difficult to determine and could change over time, as users may potentially change their behaviour as their needs evolve. The determination of the operational profile involves identifying the common operations to be performed, and the probability of each operation being performed.

Cleanroom employs *statistical usage testing* rather than coverage testing, and this involves executing tests chosen from the population of all possible uses of the software in accordance with the probability of its expected use. The software reliability measure is calculated by statistical techniques based on the expected usage of the software, and Cleanroom provides a certified mean time to failure of the software.

Coverage testing involves designing tests that cover every path through the program, and this type of testing is as likely to find a rare execution failure as well as a frequent execution failure. However, it is essential to find failures that occur on frequently used parts of the system.

The advantage of usage testing (that matches the actual execution profile of the software) is that it has a better chance of finding execution failures on frequently used parts of the system. This helps to maximize the expected mean time to failure of the software.

The Cleanroom software development process and calculation of the software reliability measure is described in [ORg:06], and the Cleanroom development process enables engineers to deliver high-quality software on time and on budget.

Table 2.3 Cleanroom results in IBM

Project	Results
Flight control project (1987) 33KLOC	Completed ahead of schedule Error-fix effort reduced by factor of five 2.5 errors KLOC before any execution
Commercial product (1988)	Deployment failures of 0.1/KLOC Certification testing failures 3.4/ KLOC Productivity 740 LOC/month
Satellite Control (1989) 80 KLOC (partial cleanroom)	50% improvement in quality Certification testing failures of 3.3/KLOC Productivity 780 LOC/month 80% improvement in productivity
Research project (1990) 12 KLOC	Certified to 0.9978 with 989 test cases

Some of the benefits of the use of Cleanroom on projects at IBM are described in [CoM:90] and summarized in Table 2.3.

2.2.3 Software Reliability Models

Models are simplifications of the reality, and a good model allows accurate predictions of future behaviour to be made. It is important to determine the adequacy of the model, and this is done by model exploration, and determining the extent to which it explains the actual manifested behaviour, as well as the accuracy of its predictions.

A model is judged effective if there is good empirical evidence to support it, and more accurate models are sought to replace inadequate models. Models are often modified (or replaced) over time, as further facts and observations lead to aberrations that cannot be explained with the current model. A good software reliability model will have the following characteristics (Table 2.4).

There are several software reliability predictor models employed (Table 2.5). Some of them just compute defect counts rather than estimating software reliability in terms of mean time to failure. They may be categorized into:

Table 2.4 Characteristics of good software reliability model

Good theoretical foundation
Realistic assumptions
Good empirical support
As simple as possible (Ockham's Razor)
Trustworthy and accurate

Table 2.5 Software reliability models

Model	Description	Comments
Jelinski/Moranda model	The failure rate is a Poisson process ^a and is proportional to the current defect content of program. The initial defect count is N ; the initial failure rate is $N\phi$; it decreases to $(N - 1)\phi$ after the first fault is detected and eliminated, and so on. The constant ϕ is termed the proportionality constant	Assumes defects are corrected perfectly, and no new defects are introduced Assumes each fault contributes the same amount to failure rate
Littlewood/Verrall model	Successive execution time between failures is independent exponentially distributed random variables ^b . Software failures are the result of the particular inputs, and faults are introduced from the correction of defects	Does not assume perfect correction of defects
Seeding and tagging	This is analogous to estimating the fish population of a lake (Mills). A known number of defects are inserted into a software program, and the proportion of these identified during testing is determined Another approach (Hyman) is to regard the defects found by one tester as tagged, and then to determine the proportion of tagged defects found by a second independent tester	Estimate of the total number of defects in the software but not a not s/w reliability predictor Assumes all faults are equally likely to be found and introduced faults representative of existing
Generalized Poisson model	The number of failures observed in i th time interval τ_i has a Poisson distribution with mean $\phi(N - M_{i-1})\tau_i^\alpha$, where N is the initial number of faults; M_{i-1} is the total number of faults removed up to the end of the $(i - 1)$ th time interval; and ϕ is the proportionality constant	Assumes faults are removed perfectly at end of time interval

^aThe Poisson process is a widely used counting process, and especially in counting the occurrence of certain events that appear to happen at a certain rate but at random. A Poisson random variable is of the form $P\{X = i\} = e^{-\lambda} \lambda^i / i!$

^bThe exponential distribution is used to model the time between the occurrence of events in an interval of time. The density function is given by $f(x) = \lambda e^{-\lambda x}$

- *Size and Complexity Metrics*
These are used to predict the number of defects that a system will reveal in operation or testing.
- *Operational Usage Profile*
These predict failure rates are based on the expected operational usage profile of the system. The number of failures encountered is determined, and the software reliability is predicted (e.g. Cleanroom and its prediction of the MTTF).
- *Quality of the Development Process*
These predict failure rates are based on the process maturity of the software development process in the organization (e.g. CMMI maturity).

The extent to which the software reliability model can be trusted depends on the accuracy of its predictions, and empirical data will need to be gathered to make a judgment. It may be acceptable to have a little inaccuracy during the early stages of prediction, provided the predictions of operational reliability are close to the observations. A model that gives overly optimistic results is termed “optimistic”, whereas a model that gives overly pessimistic results is termed “pessimistic”.

The assumptions in the reliability model need to be examined to determine whether they are realistic. Several software reliability models have questionable assumptions such as:

- All defects are corrected perfectly.
- Defects are independent of one another.
- Failure rate decreases as defects are corrected.
- Each fault contributes the same amount to the failure rate.

2.3 Dependability

Software is ubiquitous and is important to all sections of society, and so it is essential that widely used software is dependable (or trustworthy). In other words, the software should be available whenever required, as well as operating properly, safely and reliably, without any adverse side effects or security concerns. It is essential that the software used in systems in the safety critical and security critical fields is dependable, as the consequence of failure (e.g. the failure of a nuclear power plant) could be massive damage leading to loss of life or endangering the lives of the public.

Dependability engineering is concerned with techniques to improve the dependability of systems, and it involves the use of a rigorous design and development process to minimize the number of defects in the software. A dependable system is generally designed for fault tolerance, where the system can deal with (and recover from) faults that occur during software execution. Such a system needs

Table 2.6 Dimensions of dependability

Dimension	Description
Availability	System is available for use at any time
Reliability	The system operates correctly and is trustworthy
Safety	The system does not injure people or damage the environment
Security	The system prevents unauthorized intrusions

to be secure, and able to protect itself from accidental or deliberate external attacks. Table 2.6 lists several dimensions of dependability.

Modern software systems are subject to attack by malicious software such as viruses that change the behaviour of the software, or corrupt data causing the system to become unreliable. Other malicious attacks include a denial of service attack that negatively impacts the system's availability.

The design and development of dependable software needs to include protection measures that protect against external attacks that could compromise the availability and security of the system. Further, a dependable system needs to include recovery mechanisms to enable normal service to be restored as quickly as possible following an attack.

Dependability engineering is concerned with techniques to improve the dependability of systems, and in designing dependable systems. A dependable system will generally be developed using an explicitly defined repeatable process, and it may employ *redundancy* (spare capacity) and *diversity* (different types) to achieve reliability.

There is a trade-off between dependability and the performance of the system, as dependable systems will need to carry out extra checks to monitor themselves and to check for erroneous states, and to recover from faults before failure occurs. This inevitably leads to increased costs in the design and development of dependable systems.

Software availability is the percentage of the time that the software system is running, and is a measure of the uptime/downtime of the software during a particular time period. The downtime refers to a period of time when the software is unavailable for use (including planned and unplanned outages), and many companies aim to develop software that is available for use 99.999% of the time in the year (i.e. a downtime of less than 5 min per annum). This goal is known as *five nines*, and it is a common goal in the telecommunications sector.

Safety-critical systems are systems where it is essential that the system is safe for the public, and that people or the environment is not harmed in the event of system failure. These include aircraft control systems and process control systems for chemical and nuclear power plants. The failure of a safety critical system could in some situations lead to loss of life or serious economic damage.

Formal methods are discussed in Chap. 3, and they provide a precise way of specifying the requirements of the proposed system, and demonstrating (using mathematics) that key properties are satisfied in the formal specification. Further, they may be used to show that the implemented program satisfies its specification.

The use of formal methods generally leads to increased confidence in the correctness of safety critical and security critical systems.

The security of the system refers to its ability to protect itself from accidental or deliberate external attacks, which are common today since most computers are networked and connected to the Internet. There are various security threats in any networked system including threats to the confidentiality and integrity of the system and its data, and threats to the availability of the system.

Therefore, controls are required to enhance security and to ensure that attacks are unsuccessful. Encryption is one way to reduce system vulnerability, as encrypted data is unreadable to the attacker. There may be controls that detect and repel attacks, and these controls are used to monitor the system and to take action to shut down parts of the system or restrict access in the event of an attack. There may be controls that limit exposure (e.g. insurance policies and automated backup strategies) that allow recovery from the problems introduced.

It is important to have a reasonable level of security as otherwise all of the other dimensions of dependability (reliability, availability and safety) are compromised. Security loopholes may be introduced in the development of the system, and so care needs to be taken to prevent hackers from exploiting security vulnerabilities.

Risk analysis plays a key role in the specification of security and dependability requirements, and this involves identifying risks that can result in serious incidents. This leads to the generation of specific security requirements as part of the system requirements to ensure that these risks do not materialize, or if they do materialize then serious incidents will not materialize.

2.4 Computer Security

The introduction of the Internet in the early 1990s transformed the world of computing, and it led inexorably to more and more computers being connected to the Internet. This has subsequently led to an explosive growth in attacks on computers and systems, as hackers and malicious software seek to exploit known security vulnerabilities. It is therefore essential to develop secure systems that can deal with and recover from such external attacks.

Hackers will often attempt to steal confidential data and to disrupt the services being offered by a system. Security engineering is concerned with the development of systems that can prevent such malicious attacks, and recover from them. It has become an important part of software and system engineering, and software developers need to be aware of the threats facing a system, and develop solutions to eliminate them.

Hackers may probe parts of the system for weaknesses, and system vulnerabilities may lead to attackers gaining unauthorized access to the system. There is a need to conduct a risk assessment of the security threats facing a system early in the software development process, and this will lead to several security requirements for the system.

The system needs to be designed for security, as it is difficult to add security after it has been implemented. Security loopholes may be introduced in the development of the system, and so care needs to be taken to prevent these as well as preventing hackers from exploiting security vulnerabilities. There may be controls that detect and repel attacks, and these monitor the system and take appropriate action to restrict access in the event of an attack.

The choice of architecture and how the system is organized is fundamental to the security of the system, and different types of systems will require different technical solutions to provide an acceptable level of security to its users. The following guidelines for designing secure systems are described in [Som:11]:

- Security decisions should be based on the security policy.
- A security critical system should fail securely.
- A secure system should be designed for recoverability.
- A balance is needed between security and usability.
- A single point of failure should be avoided.
- A log of user actions should be maintained.
- Redundancy and diversity should be employed.
- Organization information in system into compartments.

It is important to have a reasonable level of security, as otherwise all of the other dimensions of dependability are compromised.

2.5 System Availability

System availability is the percentage of time that the software system is running without downtime, and robust systems will generally aim to achieve 5-nines availability (i.e. 99.999% availability). This is equivalent to approximately 5 min of downtime (including planned/unplanned outages) per year. The availability of a system is measured by its performance when a subsystem fails, and its ability to resume service in a state close to the original state. A fault-tolerant system continues to operate correctly (possibly at a reduced level) after some part of the system fails, and it aims to achieve 100% availability.

System availability and software reliability are related, with availability measuring the percentage of time that the system is operational, and reliability measuring the probability of failure-free operation over a period of time. The consequence of a system failure may be to freeze or crash the system, and system availability is measured by how long it takes to recover and restart after a failure. A system may be unreliable and yet have good availability metrics (fast restart after failure), or it may be highly reliable with poor availability metrics (taking a long time to recover after a failure).

Software that satisfies strict availability constraints is usually reliable. The downtime generally includes the time needed for activities such as rebooting a machine, upgrading to a new version of software, planned and unplanned outages. It is theoretically possible for software to be highly unreliable but yet to be highly available. Consider, for example, software that fails consistently for 0.5 s every day. Then, the total failure time is 183 s or approximately 3 min, and such a system would satisfy 5-nines availability. However, this scenario is highly unlikely for almost all systems, and the satisfaction of strict availability constraints usually means that the software is also highly reliable.

It is possible that software that is highly reliable may satisfy poor availability metrics. Consider the upgrade of the version of software at a customer site to a new version, where the upgrade path is complex or poorly designed (e.g. taking 2 days). Then, the availability measure is very poor even though the product may be highly reliable. Further, the time that system unavailability occurs is relevant, as a system that is unavailable at 03:00 in the morning may have minimal impacts on users. Consequently, care is required before drawing conclusions between software reliability and software availability metrics.

2.6 Safety Critical Systems

A safety critical system is a system whose failure could result in significant economic damage or loss of life. There are many examples of safety critical systems including aircraft flight control systems and missile systems. It is therefore essential to employ rigorous processes in their design and development, and testing alone is usually insufficient to verifying the correctness of a safety critical system.

The safety critical industry takes the view that any change to safety critical software creates a new program. The new program is therefore required to demonstrate that it is reliable and safe to the public, and so extensive testing needs to be performed. Other techniques such as formal verification and model checking may be employed to provide an extra level of assurance in the correctness of the safety critical system.

Safety critical systems need to be dependable and available for use whenever required. Safety critical software must operate correctly and reliably without any adverse side effects. The consequence of failure (e.g. the failure of a weapons system) could be massive damage, leading to loss of life or endangering the lives of the public.

Safety critical systems are generally designed for fault tolerance, where the system can deal with (and recover from) faults that occur during execution. Fault tolerance is achieved by anticipating exceptional events, and in designing the system to handle them. A fault-tolerant system is designed to fail safely, and programs are designed to continue working (possibly at a reduced level of performance) rather than crashing after the occurrence of an error or exception. Many fault-tolerant systems mirror all operations, where each operation is performed on

two or more duplicate systems, and so if one fails, then the other system can take over.

The development of a safety critical system needs to be rigorous, and subject to strict quality assurance to ensure that the system is safe to use and that the public will not be in danger. This involves rigorous design and development processes to minimize the number of defects in the software, as well as comprehensive testing to verify its correctness.

Formal methods consist of a set of mathematical techniques to rigorously state the requirements of the proposed system. They may be employed to derive a program from its mathematical specification, and they may be used to provide a rigorous proof that the implemented program satisfies its specification. The advantages of a mathematical specification are that it is not subject to the ambiguities inherent in a natural language description of a system, and they may be subjected to a rigorous analysis to demonstrate the presence or absence of key properties.

2.7 Review Questions

1. Explain the difference between software reliability and system availability.
2. What is software dependability?
3. Explain the significance of Adam's 1984 study of failures at IBM.
4. Describe the Cleanroom methodology.
5. Describe the characteristics of a good software reliability model.
6. Explain the relevance of security engineering.
7. What is a safety critical system?

2.8 Summary

This chapter gave an introduction to some important topics in software engineering including software reliability and the Cleanroom methodology; dependability; availability; security; and safety critical systems.

Software reliability is the probability that the program works without failure for a period of time, and it is usually expressed as the mean time to failure. Cleanroom involves the application of statistical techniques to calculate software reliability, and it is based on the expected usage of the software.

It is essential that software used in the safety and security critical fields is dependable, with the software available when required, as well as operating safely and reliably without any adverse side effects. Many of these systems are fault tolerant and are designed to deal with (and recover) from faults that occur during execution.

Such a system needs to be secure and able to protect itself from external attacks, and needs to include recovery mechanisms to enable normal service to be restored as quickly as possible. In other words, it is essential that if the system fails, then it fails safely.

Today, billions of computers are connected to the Internet, and this has led to a growth in attacks on computers. It is essential that developers are aware of the threats facing a system and are familiar with techniques to eliminate them.

3.1 Introduction

The term “*formal methods*” refers to various mathematical techniques used for the formal specification and development of software. They consist of a formal specification language and employ a collection of tools to support the syntax checking of the specification, as well as the proof of properties of the specification. They allow questions to be asked about what the system does independently of the implementation.

The use of mathematical notation avoids speculation about the meaning of phrases in an imprecisely worded natural language description of a system. Natural language is inherently ambiguous, whereas mathematics employs a precise rigorous notation. Spivey [Spi:92] defines formal specification as:

Definition 3.1 (*Formal Specification*)

Formal specification is the use of mathematical notation to describe in a precise way the properties that an information system must have, without unduly constraining the way in which these properties are achieved.

The formal specification thus becomes the key reference point for the different parties involved in the construction of the system. It may be used as the reference point for the requirements; program implementation; testing and program documentation. It thus promotes a common understanding for all those concerned with the system. The term “*formal methods*” is used to describe a formal specification language and a method for the design and implementation of a computer system. Formal methods may be employed at a number of levels:

- Formal specification only (program developed informally);
- Formal specification, refinement and verification (some proofs);
- Formal specification, refinement and verification (with extensive theorem proving).

The specification is written in a mathematical language, and the implementation may be derived from the specification via stepwise refinement.¹ The refinement step makes the specification more concrete and closer to the actual implementation. There is an associated proof obligation to demonstrate that the refinement is valid, and that the concrete state preserves the properties of the abstract state. Thus, assuming that the original specification is correct and the proof of correctness of each refinement step is valid, then there is a very high degree of confidence in the correctness of the implemented software.

Stepwise refinement is illustrated as follows: the initial specification S is the initial model M_0 ; it is then refined into the more concrete model M_1 , and M_1 is then refined into M_2 , and so on until the eventual implementation $M_n = E$ is produced.

$$S = M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq M_3 \sqsubseteq \dots \sqsubseteq M_n = E.$$

Requirements are the foundation of the system to be built, and irrespective of the best design and development practices, the product will be incorrect if the requirements are incorrect. The objective of requirements validation is to ensure that the requirements reflect what is actually required by the customer (in order to build the right system). Formal methods may be employed to model the requirements, and the model exploration yields further desirable or undesirable properties.

Formal methods provide the facility to prove that certain properties are true of the specification, and this is valuable, especially in safety critical and security critical applications. The properties are a logical consequence of the mathematical definition, and the requirements may be amended where appropriate. Thus, formal methods may be employed in a sense to debug the requirements during requirements validation.

The use of formal methods generally leads to more robust software and increased confidence in its correctness. Formal methods may be employed at different levels (e.g. just for specification with the program developed informally). The challenges involved in the deployment of formal methods in an organization include the education of staff in formal specification, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods have been applied to a diverse range of applications, including the safety and security critical fields to develop dependable software. The applications include the railway sector, microprocessor verification, the specification of standards, and the specification and verification of programs. Parnas and others have criticized formal methods (Table 3.1):

However, formal methods are potentially quite useful and reasonably easy to use. The use of a formal method such as Z or VDM forces the software engineer to be precise and helps to avoid ambiguities present in natural language. Clearly, a

¹It is questionable whether stepwise refinement is cost effective in mainstream software engineering, as it involves rewriting a specification *ad nauseum*. It is time consuming to proceed in refinement steps with significant time also required to prove that the refinement step is valid. It is more relevant to the safety-critical field. Others in the formal methods field may disagree with this position.

Table 3.1 Criticisms of formal methods

No.	Criticism
1.	Often the formal specification is as difficult to read as the program ^a
2.	Many formal specifications are wrong ^b
3.	Formal methods are strong on syntax but provide little assistance in deciding on what technical information should be recorded using the syntax ^c
4.	Formal specifications provide a model of the proposed system. However, a precise unambiguous mathematical statement of the requirements is what is needed ^d
5.	Stepwise refinement is unrealistic. It is like, for example, deriving a bridge from the description of a river and the expected traffic on the bridge. There is always a need for the creative step in design ^e
6.	Many unnecessary mathematical formalisms have been developed rather than using the available classical mathematics ^f

^aOf course, others might reply by saying that some of Parnas's tables are not exactly intuitive, and that the notation he employs in some of his tables is quite unfriendly. The usability of all of the mathematical approaches needs to be enhanced if they are to be taken seriously by industrialists

^bObviously, the formal specification must be analysed using mathematical reasoning and tools to provide confidence in its correctness. The validation of a formal specification can be carried out using mathematical proof of key properties of the specification; software inspections; or specification animation

^cApproaches such as VDM include a method for software development as well as the specification language

^dModels are extremely valuable as they allow simplification of the reality. A mathematical study of the model demonstrates whether it is a suitable representation of the system. Models allow properties of the proposed requirements to be studied prior to implementation

^eStepwise refinement involves rewriting a specification with each refinement step producing a more concrete specification (that includes code and formal specification) until eventually the detailed code is produced. It is difficult and time consuming but, tool support may make refinement easier

^fApproaches such as VDM or Z are useful in that they add greater rigour to the software development process. They are reasonably easy to learn, and there have been some good results obtained by their use. Classical mathematics is familiar to students, and therefore it is desirable that new formalisms are introduced only where absolutely necessary

formal specification should be subject to peer review to provide confidence in its correctness. New formalisms need to be intuitive to be usable by practitioners, and an advantage of classical mathematics is that it is familiar to students.

3.2 Why Should We Use Formal Methods?

There is a strong motivation to use best practice in software engineering in order to produce software adhering to high-quality standards. Quality problems with software may cause minor irritations or major damage to a customer's business including loss of life. Formal methods are a leading-edge technology that may be of benefit to companies in reducing the occurrence of defects in software products. Brown [Bro:90] argues that for the safety critical field that:

Comment 3.1 (*Missile Safety*)

Missile systems must be presumed dangerous until shown to be safe, and that the absence of evidence for the existence of dangerous errors does not amount to evidence for the absence of danger.

This suggests that companies in the safety critical field will need to demonstrate that every reasonable practice was taken to prevent the occurrence of defects. One such practice is the use of formal methods, and its exclusion may need to be justified in some domains. It is quite possible that a software company may be sued for software which injures a third party, and this suggests that companies will need a rigorous quality assurance system to prevent the occurrence of defects.

There is some evidence to suggest that the use of formal methods provides savings in the cost of the project. For example, a 9% cost saving is attributed to the use of formal methods during the CICS project; the T800 project attributes a 12-month reduction in testing time to the use of formal methods. These are discussed in more detail in Chap. 1 of [HB:95].

The use of formal methods is mandatory in certain circumstances. The Ministry of Defence (MOD) in the UK issued two safety-critical standards² in the early 1990s related to the use of formal methods in the software development life cycle.

The first is Defence Standard 00-55, “*The Procurement of safety critical software in defense equipment*” [MOD:91a], which makes it mandatory to employ formal methods in the development of safety-critical software in the UK. The standard mandates the use of formal proof that the most crucial programs correctly implement their specifications.

The other is Def. Stan 00-56 “*Hazard analysis and safety classification of the computer and programmable electronic system elements of defense equipment*” [MOD:91b]. The objective of this standard is to provide guidance to identify which systems or parts of systems being developed are safety-critical and thereby require the use of formal methods. This proposed system is subject to an initial hazard analysis to determine whether there are safety-critical parts.

The reaction to these defence standards 00-55 and 00-56 was quite hostile initially, as most suppliers were unlikely to meet the technical and organization requirements of the standard. This is described in [Tie:91].

3.3 Industrial Applications of Formal Methods

Formal methods have been employed in several domains such as the transport sector, the nuclear sector, the space sector, the defence sector, the semiconductor sector, the financial sector and the telecom sector. The extent of the application of formal methods has varied from formal specification only, to specification with

²The UK Defence Standards 0055 and 0056 were later revised to be less prescriptive on the use of formal methods.

inspections, to proofs, to refinement, to test generation and to model checking. Formal methods are applicable to the regulated sector, and it has been applied to real-time applications in the nuclear industry, the aerospace industry, the security technology area and the railroad domain. These sectors are subject to stringent regulatory controls to ensure that safety and security are properly addressed.

Several organizations have piloted formal methods with varying degrees of success. IBM developed the VDM specification language at its laboratory in Vienna, and it piloted the Z and B formal specification languages on the CICS (Customer Information Control System) project at its plant in Hursley, England.

The mathematical techniques developed by Parnas (i.e. his requirements model and tabular expressions) were employed to specify the requirements of the A-7 aircraft (as part of a research project for the US Navy).³ Tabular expressions were also employed for the software inspection of the automated shutdown software of the Darlington nuclear power plant in Canada.⁴ These are two successful uses of mathematical techniques in software engineering.

There are examples of the use of formal methods in the railway domain, with GEC Alstom and RATP using B for the formal specification and verification of the computerized signalling system on the Paris Metro. Several examples dealing with the modelling and verification of a railroad gate controller and railway signalling are described in [HB:95]. Clearly, it is essential to verify safety critical properties such as *“when the train goes through the level crossing then the gate is closed”*.

PVS is a mechanized environment for formal specification and verification, and it was developed at SRI in California. It includes a specification language integrated with support tools and an interactive theorem prover. The specification language is based on higher-order logic, and the theorem prover is guided by the user in conducting proof. It has been applied to the verification of hardware and software, and PVS has been used for the formal specification and partial verification of the microcode of the AAMP5 microprocessor.

A selection of applications of formal methods to industry is presented in [Woo:09].

3.4 Industrial Tools for Formal Methods

Formal methods have been criticized for the limited availability of tools to support the software engineer in writing the formal specification and in conducting proof. Many of the early tools were criticized as not being of industrial strength. However, in recent years more advanced tools have become available to support the software

³However, the resulting software was never actually deployed on the A-7 aircraft.

⁴This was an impressive use of mathematical techniques, and it has been acknowledged that formal methods must play an important role in future developments at Darlington. However, given the time and cost involved in the software inspection of the shutdown software, some managers have less enthusiasm in shifting from hardware to software controllers [Ger:94].

engineer's work in formal specification and formal proof, and this is likely to continue in the coming years.

The tools include syntax checkers that determine whether the specification is syntactically correct; specialized editors which ensure that the written specification is syntactically correct; tools to support refinement; automated code generators that generate a high-level language corresponding to the specification; theorem provers to demonstrate the correctness of refinement steps and to identify and resolve proof obligations, as well proving the presence or absence of key properties; and specification animation tools where the execution of the specification can be simulated.

The *B-Toolkit*⁵ from *B-Core* is an integrated set of tools that supports the *B-Method*. It provides functionality for syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assister and code generation. This, in theory, allows the complete formal development from the initial specification to the final implementation, with every proof obligation justified, leading to a provably correct program. There is also the *Atelier B* tool to support formal specification and development in *B*.

The *IFAD Toolbox*⁶ is a support tool for the *VDM-SL* specification language, and it provides support for syntax and type checking, an interpreter and debugger to execute and debug the specification, and a code generator to convert from *VDM-SL* to *C++*. The *Overture Integrated Development Environment (IDE)* is an open-source tool for formal modelling and analysis of *VDM-SL* specifications.

There are various tools for model checking including *Spin*, *Bandera*, *SMV* and *Uppaal*. These tools perform a systematic check on property *P* in all states and are applicable if the system generates a finite behavioural model. *Spin* is an open-source tool, and it checks finite-state systems with properties specified by linear temporal logic. It generates a counter-example trace if it determines that a property is violated.

There are tools to support theorem provers, and the *Boyer-Moore theorem prover (NQTHM)* was developed at the University of Texas in the late 1970s. It is far more automated than many other interactive theorem provers, but it requires detailed human guidance (with suggested lemmas) for difficult proofs. The user therefore needs to understand the proof being sought and the internals of the theorem prover. Many well known mathematical theorems have been proved with *NQTHM* including Gödel's incompleteness theorem.

The *HOL* system was developed at the University of Cambridge, and it is an environment for interactive theorem proving in a higher-order logic. It requires skilled human guidance and has been used for the verification of microprocessor design. It is one of the most widely used theorem provers.

⁵The source code for the *B-Toolkit* is now available.

⁶The *IFAD Toolbox* has been renamed to *VDM Tools* as *IFAD* sold the *VDM Tools* to *CSK* in Japan. The *CSK VDM Tools* are available for worldwide use.

3.5 Approaches to Formal Methods

There are two key approaches to formal methods: namely the *model-oriented approach* of VDM or Z, and the *algebraic* or *axiomatic approach* of the process calculi such as the calculus communicating systems (CCS) or communicating sequential processes (CSP).

3.5.1 Model-Oriented Approach

The model-oriented approach to specification is based on mathematical models, where a model is a simplification or abstraction of the real world that contains only the essential details. For example, the model of an aircraft will not include the colour of the aircraft, and the objective may be to model the aerodynamics of the aircraft. There are many models employed in the physical world, such as meteorological models that allow weather forecasts to be made.

The importance of models is that they serve to explain the behaviour of a particular entity and may also be used to predict future behaviour. Different models may vary in their ability to explain aspects of the entity under study. One model may be good at explaining some aspects of the behaviour, whereas another model might be good at explaining other aspects. The *adequacy* of a model is a key concept in modelling, and it is determined by its effectiveness in representing the underlying behaviour, and in its ability to predict future behaviour. Model exploration consists of asking questions, and determining whether the model is able to give an effective answer to the particular question. A good model is chosen as a representation of the real world and is referred to whenever there are questions in relation to the aspect of the real world.

It is fundamental to explore the model to determine its adequacy, and to determine the extent to which it explains the underlying physical behaviour, and allows accurate predictions of future behaviour to be made. There may be more than one possible model of a particular entity; for example, the Ptolemaic model and the Copernican model are different models of the solar system. This leads to the question as to which is the best or most appropriate model to use, and on the criteria to use to determine which is more suitable. The ability of the model to explain the behaviour, its simplicity and its elegance will be part of the criteria. The principle of "*Ockham's Razor*" (law of parsimony) is used in modelling, and it suggests that the simplest model with the least number of assumptions required should be selected.

The adequacy of the model will determine its acceptability as a representation of the physical world. Models that are ineffective will be replaced with models that offer a better explanation of the manifested physical behaviour. There are many examples in science of the replacement of one theory by a newer one. For example, the Copernican model of the universe replaced the older Ptolemaic model, and Newtonian physics was replaced by Einstein's theories of relativity. The structure of the revolutions that take place in science are described in [Kuh:70].

Modelling can play a key role in computer science, as computer systems tend to be highly complex, whereas a model allows simplification or an abstraction of the underlying complexity, and it enables a richer understanding of the underlying reality to be gained. The model-oriented approach to software development involves defining an abstract model of the proposed software system, and the model is then explored to determine its suitability as a representation of the system. This takes the form of model interrogation, i.e. asking questions, and determining the extent to which the model can answer the questions. The modelling in formal methods is typically performed via elementary discrete mathematics, including set theory, sequences, functions and relations.

Various models have been applied to assist with the complexities in software development. These include the Capability Maturity Model (CMM), which is employed as a framework to enhance the capability of the organization in software development; UML, which has various graphical diagrams that are employed to model the requirements and design; and mathematical models that are employed for formal specification.

VDM and Z are model-oriented approaches to formal methods. VDM arose from work done at the IBM laboratory in Vienna in formalizing the semantics for the PL/1 compiler in the early 1970s, and it was later applied to the specification of software systems. The origin of the Z specification language is in work done at Oxford University in the early 1980s.

3.5.2 Axiomatic Approach

The axiomatic approach focuses on the properties that the proposed system is to satisfy, and there is no intention to produce an abstract model of the system. The required properties and behaviour of the system are stated in mathematical notation. The difference between the axiomatic specification and a model-based approach may be seen in the example of a stack.

The stack includes operators for pushing an element onto the stack and popping an element from the stack. The properties of *pop* and *push* are explicitly defined in the axiomatic approach. The model-oriented approach constructs an explicit model of the stack, and the operations are defined in terms of the effect that they have on the model. The axiomatic specification of the *pop* operation on a stack is given by properties, for example, $pop(push(s, x)) = s$.

Comment 3.2 (*Axiomatic Approach*)

The property-oriented approach has the advantage that the implementer is not constrained to a particular choice of implementation, and the only constraint is that the implementation must satisfy the stipulated properties.

The emphasis is on specifying the required properties of the system, and implementation issues are avoided. The properties are typically stated using

mathematical logic or higher-order logics. Mechanized theorem-proving techniques may be employed to prove results.

One potential problem with the axiomatic approach is that the properties specified may not be satisfied in any implementation. Thus, whenever a “formal axiomatic theory” is developed a corresponding “model” of the theory must be identified, in order to ensure that the properties may be realized in practice. That is, when proposing a system that is to satisfy some set of properties, there is a need to prove that there is at least one system that will satisfy the set of properties.

3.6 Proof and Formal Methods

A mathematical proof typically includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. The proof may employ a “*divide and conquer*” technique, i.e. breaking the conjecture down into subgoals and then attempting to prove each of the subgoals.

Many proofs in formal methods are concerned with cross-checking the details of the specification, or checking the validity of the refinement steps, or checking that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers⁷ are essential in dealing with this. Machine proof is explicit and reliance on some brilliant insight is avoided. Proofs by hand are notorious for containing errors or jumps in reasoning, while machine proofs are explicit but are often extremely lengthy and unreadable. The infamous machine proof of the correctness of the VIPER microprocessor⁸ consisted of several million formulae [Tie:91].

A formal mathematical proof consists of a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules.

The application of formal methods in an industrial environment requires the use of machine-assisted proof, since thousands of proof obligations arise from a formal specification, and theorem provers are essential in resolving these efficiently. Automated theorem proving is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness⁹ is unlikely except for the most trivial of programs. A program may consist of legacy software that is assumed

⁷Most existing theorem provers are difficult to use and are for specialist use only. There is a need to improve the usability of theorem provers.

⁸This verification was controversial with RSRE and Charter overselling VIPER as a chip design that conforms to its formal specification.

⁹This position is controversial with others arguing that if correctness is defined mathematically, then the mathematical definition (i.e., formal specification) is a theorem, and the task is to prove that the program satisfies the theorem. They argue that the proofs for non-trivial programs exist,

to work; a compiler that is assumed to work correctly creates it. Theorem provers are programs that are assumed to function correctly. The best that formal methods can claim is increased confidence in correctness of the software, rather than an absolute proof of correctness.

3.7 Mathematics in Software Engineering

The debate concerning the level of use of mathematics in software engineering is still ongoing. Many practitioners are against the use of mathematics and avoid its use. They tend to employ methodologies such as software inspections and testing to improve confidence in the correctness of the software. They argue that in the current competitive industrial environment where time to market is a key driver that the use of such formal mathematical techniques would seriously impact the market opportunity. Industrialists often need to balance conflicting needs such as quality, cost and delivering on time. They argue that the commercial necessities require methodologies and techniques that allow them to achieve their business goals effectively.

The other camp argues that the use of mathematics is essential in the delivery of high-quality and reliable software, and that if a company does not place sufficient emphasis on quality, it will pay the price in terms of poor quality and loss of reputation.

It is generally accepted that mathematics and formal methods must play a role in the safety critical and security critical fields. Apart from that, the extent of the use of mathematics is a hotly disputed topic. The pace of change in the world is extraordinary, and companies face significant competitive forces in a global marketplace.

It is unrealistic to expect companies to deploy formal methods unless they have clear evidence that it will support them in delivering commercial products to the marketplace ahead of their competition, at the right price and with the right quality. Formal methods need to prove that it can do this if it wishes to be taken seriously in mainstream software engineering. The issue of technology transfer of formal methods to industry is discussed in Chap. 18.

3.8 The Vienna Development Method

VDM was developed by a research team at the IBM research laboratory in Vienna. This group was specifying the semantics of the PL/I programming language using an operational semantic approach. That is, the semantics of the language were defined in terms of a hypothetical machine which interprets the programs of that

and that the reason why there are not many examples of such proofs is due to a lack of mathematical specifications.

language [BjJ:78, BjJ:82]. Later work led to the Vienna Development Method (VDM) with its specification language, Meta IV. This was used to give the denotational semantics of programming languages; that is, a mathematical object (set, function, etc.) is associated with each phrase of the language. The mathematical object is termed the *denotation* of the phrase.

VDM is a *model-oriented approach* and this means that an explicit model of the state of an abstract machine is given, and operations are defined in terms of the state. Operations may act on the system state, taking inputs and producing outputs as well as a new system state. Operations are defined in a precondition and postcondition style. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant.

VDM uses keywords to distinguish different parts of the specification, e.g. preconditions, postconditions, as introduced by the keywords *pre* and *post*, respectively. In keeping with the philosophy that formal methods specify *what* a system does as distinct from *how*, VDM employs postconditions to stipulate the effect of the operation on the state. The previous state is then distinguished by employing *hooked variables*, e.g. v^- and the postcondition specifies the new state which is defined by a logical predicate relating the prestate to the poststate.

VDM is more than its specification language VDM-SL and is, in fact, a software development method, with rules to verify the steps of development. The rules enable the executable specification, i.e. the detailed code, to be obtained from the initial specification via refinement steps. Thus, we have a sequence $S = S_0, S_1, \dots, S_n = E$ of specifications, where S is the initial specification and E is the final (executable) specification.

Retrieval functions enable a return from a more concrete specification to the more abstract specification. The initial specification consists of an initial state, a system state and a set of operations. The system state is a particular domain, where a domain is built out of primitive domains such as the set of natural numbers, integers or constructed from primitive domains using domain constructors such as Cartesian product, disjoint union. A domain-invariant predicate may further constrain the domain, and a *type* in VDM reflects a domain obtained in this way. Thus, a type in VDM is more specific than the signature of the type and thus represents values in the domain defined by the signature, which satisfy the domain invariant. In view of this approach to types, it is clear that VDM types may not be “statically type checked”.

VDM specifications are structured into modules, with a module containing the module name, parameters, types, operations, etc. Partial functions occur frequently in computer science as many functions, may be undefined, or fail to terminate for some arguments in their domain. VDM addresses partial functions by employing non-standard logical operators, namely the logic of partial functions (LPFs), which is discussed in Chap. 7.

VDM has been used in industrial projects, and its tool support includes the IFAD Toolbox.¹⁰ VDM is described in more detail in Chap. 9. There are several variants of VDM, including VDM++, the object-oriented extension of VDM and the Irish School of the VDM, which is discussed in the next section.

3.9 VDM^{*}, the Irish School of VDM

The Irish School of VDM is a variant of standard VDM and is characterized by its constructive approach, classical mathematical style and its terse notation [Mac:90]. This method aims to combine the *what* and *how* of formal methods in that its terse specification style stipulates in concise form *what* the system should do; furthermore, the fact that its specifications are constructive (or functional) means that the *how* is included with the *what*.

However, it is important to qualify this by stating that the *how* as presented by VDM^{*} is not directly executable, as several of its mathematical data types have no corresponding structure in high-level programming languages or functional languages. Thus, a conversion or reification of the specification into a functional or higher-level language must take place to ensure a successful execution. Further, the fact that a specification is constructive is no guarantee that it is a good implementation strategy if the construction itself is naive.

The Irish school follows a similar development methodology as in standard VDM, and it is a model-oriented approach. The initial specification is presented, with the initial state and operations defined. The operations are presented with preconditions; however, no postcondition is necessary as the operation is “functionally” (i.e. explicitly) constructed.

There are proof obligations to demonstrate that the operations preserve the invariant. That is, if the precondition for the operation is true, and the operation is performed, then the system invariant remains true after the operation. The philosophy is to exhibit existence *constructively* rather than providing a theoretical proof of existence that demonstrates the existence of a solution without presenting an algorithm to construct the solution.

The school avoids the existential quantifier of predicate calculus, and reliance on logic in proof is kept to a minimum, with emphasis instead placed on equational reasoning. Structures with nice algebraic properties are sought, and one nice algebraic structure employed is the monoid, which has closure, associative and a unit element. The concept of isomorphism is powerful, reflecting that two structures are essentially identical, and thus we may choose to work with either, depending on which is more convenient for the task in hand.

The school has been influenced by the work of Polya and Lakatos. The former [Pol:57] advocated a style of problem-solving characterized by first considering an easier subproblem, and considering several examples. This generally leads to a

¹⁰The VDM Tools are now available from the CSK Group in Japan.

clearer insight into solving the main problem. Lakatos's approach to mathematical discovery [Lak:76] is characterized by heuristic methods. A primitive conjecture is proposed, and if global counter-examples to the statement of the conjecture are discovered, then the corresponding *hidden lemma* for which this global counter-example is a local counter-example is identified and added to the statement of the primitive conjecture. The process repeats, until no more global counter-examples are found. A sceptical view of absolute truth or certainty is inherent in this.

Partial functions are the norm in VDM⁺, and as in standard VDM, the problem is that functions may be undefined, or fail to terminate for several of the arguments in their domain. The logic of partial functions (LPFs) is avoided, and instead care is taken with recursive definitions to ensure termination is achieved for each argument. Academic and industrial projects have been conducted using VDM⁺, but tool support is limited. The Irish School of VDM is discussed in more detail in Chap. 10.

3.10 The Z Specification Language

Z is a formal specification language founded on Zermelo set theory, and it was developed by Abrial at Oxford University in the early 1980s. It is used for the formal specification of software and is a model-oriented approach. An explicit model of the state of an abstract machine is given, and the operations are defined in terms of the effect on the state. It includes a mathematical notation that is similar to VDM and the visually striking schema calculus. The latter consists essentially of boxes (or schemas), and these are used to describe operations and states. The schema calculus enables schemas to be used as building blocks and combined with other schemas. The Z specification language was published as an ISO standard (ISO/IEC 13568:2002) in 2002.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make Z specification highly readable, as each individual schema is small in size and self-contained. Exception handling is done by defining schemas for the exception cases, and these are then combined with the original operation schema. Mathematical data types are used to model the data in a system, and these data types obey mathematical laws. These laws enable simplification of expressions and are useful with proofs.

Operations are defined in a precondition/postcondition style. However, the precondition is implicitly defined within the operation; that is, it is not separated out as in standard VDM. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant. Postconditions employ a logical predicate which relates the prestate to the poststate, and the poststate of a variable v is given by priming, e.g. v' . Various conventions are employed, e.g. $v?$ indicates that v is an input variable and $v!$ indicates that v is an

output variable. The symbol ΞOp operation indicates that this operation does not affect the state, whereas ΔOp indicates that this operation affects the state.

Many data types employed in Z have no counterpart in standard programming languages. It is, therefore, important to identify and describe the concrete data structures that will ultimately represent the abstract mathematical structures. The operations on the abstract data structures may need to be refined to yield operations on the concrete data structure that yield equivalent results. For simple systems, direct refinement (i.e. one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement is employed, where a sequence of increasingly concrete specifications are produced to eventually yield the executable specification.

Z has been successfully applied in industry, and one of its well-known successes is the CICS project at IBM Hursley in England. Z is described in more detail in Chap. 8.

3.11 The B -Method

The *B-Technologies* [McD:94] consist of three components: a method for software development, namely the B -Method; a supporting set of tools, namely the B -Toolkit; and a generic program for symbol manipulation, namely the B -Tool (from which the B -Toolkit is derived). The B -Method is a model-oriented approach and is closely related to the Z specification language. Abrial developed the B specification language, and every construct in the language has a set theoretic counterpart, and the method is founded on Zermelo set theory. Each operation has an explicit precondition.

A key role of the *abstract machine* in the B -Method is to provide encapsulation of variables representing the state of the machine and operations that manipulate the state. Machines may refer to other machines, and a machine may be introduced as a refinement of another machine. The abstract machines are specification machines, refinement machines or implementable machines. The B -Method adopts a layered approach to design where the design is gradually made more concrete by a sequence of design layers. Each design layer is a refinement that involves a more detailed implementation in terms of the abstract machines of the previous layer. The design refinement ends when the final layer is implemented purely in terms of library machines. Any refinement of a machine by another has associated proof obligations, and proof is required to verify the validity of the refinement step.

Specification animation of the abstract machine notation (AMN) specification is possible with the B -Toolkit, and this enables typical usage scenarios to be explored for requirements validation. This is, in effect, an early form of testing, and it may be used to demonstrate the presence or absence of desirable or undesirable behaviour. Verification takes the form of a proof to demonstrate that the invariant is preserved when the operation is executed within its precondition, and this is performed on the AMN specification with the B -Toolkit.

The *B*-Toolkit provides several tools that support the *B*-Method, and these include syntax and type checking, specification animation, proof obligation generator, auto-prover, proof assistor and code generation. Thus, in theory, a complete formal development from initial specification to final implementation may be achieved, with every proof obligation justified, leading to a provably correct program.

The *B*-Method and toolkit have been successfully applied in industrial applications, including the CICS project at IBM Hursley in the UK [Hoa:95]. The automated support provided has been cited as a major benefit of the application of the *B*-Method and the *B*-Toolkit.

3.12 Predicate Transformers and Weakest Preconditions

The precondition of a program S is a predicate, i.e. a statement that may be true or false, and it is usually required to prove that if the precondition Q is true, then execution of S is guaranteed to terminate in a finite amount of time in a state satisfying R . This is written as $\{Q\} S \{R\}$.

The weakest precondition of a command S with respect to a postcondition R [Gri:81] represents the set of all states such that if execution begins in any one of these states, then execution will terminate in a finite amount of time in a state with R true. These set of states may be represented by a predicate Q' , so that $wp(S, R) = wp_S(R) = Q'$, and so wp_S is a predicate transformer; that is, it may be regarded as a function on predicates. The weakest precondition is the precondition that places the fewest constraints on the state than all of the other preconditions of (S, R) . That is, all of the other preconditions are stronger than the weakest precondition.

The notation $Q\{S\}R$ is used to denote partial correctness, and indicates that if execution of S commences in any state satisfying Q , and if execution terminates, then the final state will satisfy R . Often, a predicate Q which is stronger than the weakest precondition $wp(S, R)$ is employed, especially where the calculation of the weakest precondition is non-trivial. Thus, a stronger predicate Q such that $Q \Rightarrow wp(S, R)$ is often employed.

There are many properties associated with the weakest preconditions, and these may be used to simplify expressions involving weakest preconditions, and in determining the weakest preconditions of various program commands such as assignments, iterations. Weakest preconditions may be used in developing a proof of correctness of a program in parallel with its development [ORg:06].

An imperative program may be regarded as a predicate transformer. This is since a predicate P characterises the set of states in which the predicate P is true, and an imperative program may be regarded as a binary relation on states, which, leads to the Hoare triple $P\{F\}Q$. That is, the program F acts as a predicate transformer with the predicate P regarded as an input assertion, i.e. a Boolean expression that must be true before the program F is executed, and the predicate Q is the output assertion,

which is true if the program F terminates (where F commenced in a state satisfying P).

3.13 The Process Calculi

The objectives of the process calculi [Hor:85] are to provide mathematical models which provide insight into the diverse issues involved in the specification, design and implementation of computer systems which continuously act and interact with their environment. These systems may be decomposed into subsystems that interact with each other and their environment.

The basic building block is the *process*, which is a mathematical abstraction of the interactions between a system and its environment. A process that lasts indefinitely may be specified recursively. Processes may be assembled into systems; they may execute concurrently or communicate with each other. Process communication may be synchronized, and this takes the form of one process outputting a message simultaneously to another process inputting a message. Resources may be shared among several processes. Process calculi such as CSP [Hor:85] and CCS [Mil:89] have been developed, they enrich the understanding of communication and concurrency and they obey several mathematical laws.

The expression $(a ? P)$ in CSP describes a process which first engages in event a and then behaves as process P . A recursive definition is written as $(\mu X) \cdot F(X)$, and an example of a simple chocolate vending machine is:

$$\text{VMS} = \mu X: \{\text{coin}, \text{choc}\} \cdot (\text{coin} ? (\text{choc} ? X)).$$

The simple vending machine has an alphabet of two symbols, namely *coin* and *choc*. The behaviour of the machine is that a coin is entered into the machine, and then a chocolate selected and provided, and the machine is ready for further use. CSP processes use channels to communicate values with their environment, and input on channel c is denoted by $(c?.x P_x)$. This describes a process that accepts any value x on channel c and then behaves as process P_x . In contrast, $(c!e P)$ defines a process which outputs the expression e on channel c and then behaves as process P .

The π -calculus is a process calculus based on names. Communication between processes takes place between known channels, and the name of a channel may be passed over a channel. There is no distinction between channel names and data values in the π -calculus. The output of a value v on channel a is given by $\bar{a}v$; that is output is a negative prefix. Input on a channel a is given by $a(x)$ and is a positive prefix. Private links or restrictions are denoted by $(x)P$.

3.14 Finite-State Machines

Warren McCulloch and Walter Pitts published early work on finite-state automata in 1943. They were interested in modelling the thought process for humans and machines. Moore and Mealy developed this work further, and these machines are referred to as the “*Moore machine*” and the “*Mealy machine*”. The Mealy machine determines its outputs through the current state and the input, whereas the output of Moore’s machine is based upon the current state alone.

Definition 3.2 (*Finite-State Machine*)

A finite state machine (FSM) is an abstract mathematical machine that consists of a finite number of states. It includes a start state q_0 in which the machine is in initially; a finite set of states Q ; an input alphabet Σ ; a state transition function δ ; and a set of final accepting states F (where $F \subseteq Q$).

The state transition function takes the current state and an input and returns the next state. That is, the transition function is of the form:

$$\delta : Q \times \Sigma \rightarrow Q.$$

The transition function provides rules that define the action of the machine for each input, and it may be extended to provide output as well as a state transition. State diagrams are used to represent finite-state machines, and each state accepts a finite number of inputs. A finite-state machine may be deterministic or non-deterministic, and a *deterministic machine* (Fig. 3.1) changes to exactly one state for each input transition, whereas a *non-deterministic machine* may have a choice of states to move to for a particular input.

Finite-state automata can compute only very primitive functions and are not an adequate model for computing. There are more powerful automata such as the *Turing machine* that is essentially a finite automaton with an infinite storage (memory). Anything that is computable is computable by a Turing Machine.

The memory of the Turing machine is a tape that consists of a potentially infinite number of one-dimensional cells. The Turing machine provides a mathematical abstraction of computer execution and storage, as well as providing a mathematical definition of an algorithm. Automata are discussed in more detail in Chap. 13.

3.15 The Parnas Way

Parnas has been influential in the computing field, and his ideas on the specification, design, implementation, maintenance and documentation of computer software remain important. He advocates a solid engineering approach and argues that the role of the engineer is to apply scientific principles and mathematics to design and develop products. He argues that computer scientists need to be educated as

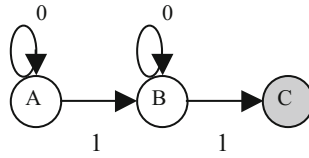


Fig. 3.1 Deterministic finite-state machine

engineers to ensure that they have the appropriate background to build software correctly.

His tabular expressions were used for the specification of the requirements of the A-7 aircraft for the US Navy, and his mathematical inspections were used to verify the correctness of the shutdown software at the Darlington nuclear power plant in Canada. Some of his contributions to software engineering are listed in (Table 3.2).

3.16 Model Checking

Model checking is an automated technique such that given a finite-state model of a system and a formal property, (expressed in temporal logic) then it systematically checks whether the property is true or false in a given state in the model. It is an effective technique to identify potential design errors, and it increases the confidence in the correctness of the system design. Model checking is a highly effective verification technology and is widely used in the hardware and software fields. It has been employed in the verification of microprocessors; in security protocols; in the transportation sector (trains); and in the verification of software in the space sector.

Model checking is a formal verification technique based on graph algorithms and formal logic. It allows the desired behaviour (specification) of a system to be verified, and its approach is to employ a suitable model of the system, and to carry out a systematic and exhaustive inspection of all states of the model to verify that the desired properties are satisfied. These properties are generally safety properties such as the absence of deadlock, request-response properties and invariants. Its systematic search determines whether a given system model truly satisfies a particular property or not. Model checking is discussed in more detail in Chap. 14.

3.17 Usability of Formal Methods

There are practical difficulties associated with the industrial use of formal methods. It seems to be assumed that programmers and customers are willing to become familiar with the mathematics used in formal methods, but this is true in only some

Table 3.2 Parnas's contributions to software engineering

Area	Contribution
Tabular expressions	These are mathematical tables for specifying requirements and enable complex predicate logic expressions to be represented in a simpler form
Mathematical documentation	He advocates the use of precise mathematical documentation for requirements and design
Requirements specification	He advocates the use of mathematical relations to specify the requirements precisely
Software design	He developed <i>information hiding</i> that is used in object-oriented design ^a and allows software to be designed for change. Every information-hiding module has an interface that provides the only means to access the services provided by the modules. The interface hides the module's implementation
Software inspections	His approach requires the reviewers to take an active part in the inspection. They are provided with a list of questions by the author, and their analysis involves the production of mathematical table to justify the answers
Predicate logic	He developed an extension of the predicate calculus to deal with partial functions, and it preserves the classical two-valued logic when dealing with undefined values

^aIt is surprising that many in the object-oriented world seem unaware that information hiding goes back to the early 1970s and many have never heard of Parnas

domains.¹¹ It is usually possible to get a developer to learn a formal method, as a programmer has some experience of mathematics and logic. However, it is more difficult to get a customer to learn a formal method, and this makes it more difficult to perform a rigorous validation of the formal specification.

This often means that often a formal specification of the requirements and an informal definition of the requirements using a natural language are maintained. It is essential that both of these are consistent and that there is a rigorous validation of the formal specification. Otherwise, if the programmer proves the correctness of the code with respect to the formal specification, and the formal specification is incorrect, then the formal development of the software is incorrect. There are several techniques to validate a formal specification including:

- Proof that the formal specification satisfies key properties;
- Software inspections to compare formal specification and informal set of requirements;
- Specification animation to validate the formal specification.

¹¹The domain in which the software is being used will influence the willingness or otherwise of the customers to become familiar with the mathematics required. There appears to be little interest in mainstream software engineering, and their perception is that formal methods are unusable. However, there is a greater interest in the mathematical approach in the safety critical field.

Formal methods are perceived as being difficult to use, and of providing limited value in mainstream software engineering. Programmers receive education in mathematics as part of their studies, but many never use formal methods again once they take an industrial position. Some of the reasons for this are:

- The notation is not intuitive.
- It is difficult to write a formal specification.
- Validation of a formal specification is difficult.
- Refinement and proof are difficult.
- Limited tool support.

It is important to investigate ways by which formal methods can be made more usable to software engineers, and technology transfer of formal methods to industry is discussed in Chap. 18. This may involve designing more usable notations and better tools to support the process. Practical training and coaching to employees can help. Some of the characteristics of a usable formal method are:

- A formal method should be intuitive.
- It should have tool support.
- A formal method should be teachable.
- It should be able to adapt to change.
- The technology transfer path should be defined.
- A formal method should be cost effective.

3.18 Review Questions

1. What are formal methods and describe their potential benefits? How essential is tool support?
2. What is stepwise refinement and how realistic is it in mainstream software engineering?
3. Discuss Parnas's criticisms of formal methods and discuss whether his views are valid.
4. Discuss the industrial applications of formal methods and which areas have benefited most from their use? What problems have arisen?
5. Describe a technology transfer path for the deployment of formal methods in an organization.
6. Explain the difference between the model-oriented approach and the axiomatic approach.

7. Discuss the nature of proof in formal methods and tools to support proof.
8. Discuss the Vienna Development Method and explain the difference between standard VDM and VDM⁺.
9. Discuss Z and B. Describe the tools in the B-Toolkit.
10. Discuss process calculi such as CSP, CCS or π -calculus.

3.19 Summary

This chapter discussed formal methods that offer a mathematical approach to the development of high-quality software. Formal methods employ mathematical techniques for the specification and development of software, and are useful in the safety critical field. They consist of a formal specification language; a methodology for formal software development; and a set of tools to support the syntax checking of the specification, as well as the proof of properties of the specification.

Formal methods may be model-oriented or axiomatic-oriented. The model-oriented approach includes formal methods such as VDM, Z and B. The axiomatic approach includes the process calculi such as CSP, CCS and the π calculus. VDM was developed at the IBM laboratory in Vienna, and has been used in academia and industry. CSP was developed by C.A.R Hoare and CCS by Robin Milner.

Formal methods allow questions to be asked and answered about what the system does independently of the implementation. They offer a way to debug the requirements and to show that certain desirable properties are true of the specification, whereas certain undesirable properties are absent.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. There are challenges involved in the deployment of formal methods, as the use of these mathematical techniques may be a culture shock to many staff.

The usability of existing formal methods was considered, and reasons for their perceived difficulty were considered. The characteristics of a usable formal method were explored.

There are various tools to support formal methods including syntax checkers; specialized editors; tools to support refinement; automated code generators that generate a high-level language corresponding to the specification; theorem provers; and specification animation tools where the execution of the specification can be simulated.

4.1 Introduction

This chapter provides an introduction to fundamental building blocks in mathematics such as sets, relations and functions. Sets are collections of well-defined objects; relations indicate relationships between members of two sets A and B ; and functions are a special type of relation where there is exactly (or at most)¹ one relationship for each element $a \in A$ with an element in B .

A set is a collection of well-defined objects that contains no duplicates. The term “well defined” means that for a given value it is possible to determine whether or not it is a member of the set. There are many examples of sets such as the set of natural numbers \mathbb{N} , the set of integer numbers \mathbb{Z} and the set of rational numbers \mathbb{Q} . The natural numbers \mathbb{N} are an infinite set consisting of the numbers $\{1, 2, \dots\}$. Venn diagrams may be used to represent sets pictorially.

A binary relation $R(A, B)$ where A and B are sets is a subset of the Cartesian product $(A \times B)$ of A and B . The domain of the relation is A , and the co-domain of the relation is B . The notation aRb signifies that there is a relation between a and b and that $(a, b) \in R$. An n -ary relation $R(A_1, A_2, \dots, A_n)$ is a subset of $(A_1 \times A_2 \times \dots \times A_n)$. However, an n -ary relation may also be regarded as a binary relation $R(A, B)$ with $A = A_1 \times A_2 \times \dots \times A_{n-1}$ and $B = A_n$.

Functions may be total or partial. A total function $f: A \rightarrow B$ is a special relation such that for each element $a \in A$ there is exactly one element $b \in B$. This is written as $f(a) = b$. A partial function differs from a total function in that the function may be undefined for one or more values of A . The domain of a function (denoted by $\mathbf{dom} f$) is the set of values in A for which the partial function is defined. The domain of the function is A provided that f is a total function. The co-domain of the function is B .

¹We distinguish between total and partial functions. A total function $f: A \rightarrow B$ is defined for every element in A , whereas a partial function may be undefined for one or more values in A .

4.2 Set Theory

A set is a fundamental building block in mathematics, and it is defined as a collection of well-defined objects. The elements in a set are of the same kind, and they are distinct with no repetition of the same element in the set.² Most sets encountered in computer science are finite, as computers can only deal with finite entities. Venn diagrams³ are often employed to give a pictorial representation of a set, and they may be used to illustrate various set operations such as set union, intersection and set difference.

There are many well-known examples of sets including the set of natural numbers denoted by \mathbb{N} ; the set of integers denoted by \mathbb{Z} ; the set of rational numbers is denoted by \mathbb{Q} ; the set of real numbers denoted by \mathbb{R} ; and the set of complex numbers denoted by \mathbb{C} .

Example 4.1 The following are examples of sets.

- The books on the shelves in a library;
- The books those are currently overdue from the library;
- The customers of a bank
- The bank accounts in a bank;
- The set of natural numbers $\mathbb{N} = \{1, 2, 3, \dots\}$;
- The integer numbers $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$;
- The non-negative integers $\mathbb{Z}^+ = \{0, 1, 2, 3, \dots\}$;
- The set of prime numbers = $\{2, 3, 5, 7, 11, 13, 17, \dots\}$;
- The rational numbers are the set of quotients of integers.

$$\mathbb{Q} = \{p/q : p, q \in \mathbb{Z} \text{ and } q \neq 0\}.$$

A finite set may be defined by listing all of its elements. For example, the set $A = \{2, 4, 6, 8, 10\}$ is the set of all even natural numbers less than or equal to 10. The order in which the elements are listed is not relevant; that is, the set $\{2, 4, 6, 8, 10\}$ is the same as the set $\{8, 4, 2, 10, 6\}$.



²There are mathematical objects known as multi-sets or bags that allow duplication of elements. For example, a bag of marbles may contain three green marbles, two blue and one red marble.

³The British logician, John Venn, invented the Venn diagram. It provides a visual representation of a set and the various set theoretical operations. Their use is limited to the representation of two or three sets as they become cumbersome with a larger number of sets.

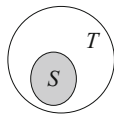
Sets may be defined by using a predicate to constrain set membership. For example, the set $S = \{n \in \mathbb{N} : n \leq 10 \wedge n \bmod 2 = 0\}$ also represents the set $\{2, 4, 6, 8, 10\}$. That is, the use of a predicate allows a new set to be created from an existing set by using the predicate to restrict membership of the set. The set of even natural numbers may be defined by a predicate over the set of natural numbers that restricts membership to the even numbers. It is defined by:

$$\text{Evens} = \{x \mid x \in \mathbb{N} \wedge \text{even}(x)\}.$$

In this example, $\text{even}(x)$ is a predicate that is true if x is even and false otherwise. In general, $A = \{x \in E \mid P(x)\}$ denotes a set A formed from a set E using the predicate P to restrict membership of A to those elements of E for which the predicate is true.

The elements of a finite set S are denoted by $\{x_1, x_2, \dots, x_n\}$. The expression $x \in S$ denotes that the element x is a member of the set S , whereas the expression $x \notin S$ indicates that x is not a member of the set S .

A set S is a subset of a set T (denoted $S \subseteq T$) if whenever $s \in S$ then $s \in T$, and in this case the set T is said to be a superset of S (denoted $T \supseteq S$). Two sets S and T are said to be equal if they contain identical elements; that is, $S = T$ if and only if $S \subseteq T$ and $T \subseteq S$. A set S is a proper subset of a set T (denoted $S \subset T$) if $S \subseteq T$ and $S \neq T$. That is, every element of S is an element of T , and there is at least one element in T that is not an element of S . In this case, T is a proper superset of S (denoted $T \supset S$).



The empty set (denoted by \emptyset or $\{\}$) represents the set that has no elements. Clearly, \emptyset is a subset of every set. The singleton set containing just one element x is denoted by $\{x\}$, and clearly $x \in \{x\}$ and $x \neq \{x\}$. Clearly, $y \in \{x\}$ if and only if $x = y$.

Example 4.2

- (i) $\{1, 2\} \subseteq \{1, 2, 3\}$;
- (ii) $\emptyset \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$;

The cardinality (or size) of a finite set S defines the number of elements present in the set. It is denoted by $|S|$. The cardinality of an infinite⁴ set S is written as $|S| = \infty$.

⁴The natural numbers, integers and rational numbers are countable sets, whereas the real and complex numbers are uncountable sets.

Example 4.3

- (i) Given $A = \{2, 4, 5, 8, 10\}$, then $|A| = 5$.
- (ii) Given $A = \{x \in \mathbb{Z} : x^2 = 9\}$, then $|A| = 2$.
- (iii) Given $A = \{x \in \mathbb{Z} : x^2 = -9\}$, then $|A| = 0$.

4.2.1 Set Theoretical Operations

Several set theoretical operations are considered in this section. These include the Cartesian product operation; the power set of a set; the set union operation; the set intersection operation; the set difference operation; and the symmetric difference operation.

Cartesian Product

The Cartesian product allows a new set to be created from existing sets. The Cartesian⁵ product of two sets S and T (denoted $S \times T$) is the set of ordered pairs $\{(s, t) \mid s \in S, t \in T\}$. Clearly, $S \times T \neq T \times S$ and so the Cartesian product of two sets is not commutative. Two ordered pairs (s_1, t_1) and (s_2, t_2) are considered equal if and only if $s_1 = s_2$ and $t_1 = t_2$.

The Cartesian product may be extended to that of n sets S_1, S_2, \dots, S_n . The Cartesian product $S_1 \times S_2 \times \dots \times S_n$ is the set of ordered tuples $\{(s_1, s_2, \dots, s_n) \mid s_1 \in S_1, s_2 \in S_2, \dots, s_n \in S_n\}$. Two ordered n -tuples (s_1, s_2, \dots, s_n) and $(s_1', s_2', \dots, s_n')$ are considered equal if and only if $s_1 = s_1', s_2 = s_2', \dots, s_n = s_n'$.

The Cartesian product may also be applied to a single set S to create ordered n -tuples of S ; that is, $S^n = S \times S \times \dots \times S$ (n times).

Power Set

The power set of a set A (denoted $\mathbb{P}A$) denotes the set of subsets of A . For example, the power set of the set $A = \{1, 2, 3\}$ has eight elements and is given by:

$$\mathbb{P}A = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

There are $2^3 = 8$ elements in the power set of $A = \{1, 2, 3\}$, and the cardinality of A is 3. In general, there are $2^{|A|}$ elements in the power set of A .

Theorem 4.1 (Cardinality of Power Set of A)

There are $2^{|A|}$ elements in the power set of A .

⁵Cartesian product is named after René Descartes who was a famous seventeenth French mathematician and philosopher. He invented the Cartesian coordinates system that links geometry and algebra, and allows geometric shapes to be defined by algebraic equations.

Proof Let $|A| = n$, then the cardinality of the subsets of A is subsets of size 0, 1, ..., n . There are $\binom{n}{k}$ subsets of A of size k .⁶ Therefore, the total number of subsets of A is the total number of subsets of size 0, 1, 2, ... up to n . That is,

$$|\mathbb{P}A| = \sum_{k=0}^n \binom{n}{k}.$$

The binomial theorem states that:

$$(1+x)^n = \sum_{k=0}^n \binom{n}{k} x^k.$$

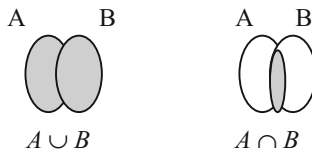
Therefore, putting $x = 1$ we get that

$$2^n = (1+1)^n = \sum_{k=0}^n \binom{n}{k} 1^k = |\mathbb{P}A|.$$

Union and Intersection Operations The union of two sets A and B is denoted by $A \cup B$. It results in a set that contains all of the members of A and of B and is defined by:

$$A \cup B = \{r \mid r \in A \text{ or } r \in B\}.$$

For example, suppose $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, then $A \cup B = \{1, 2, 3, 4\}$. Set union is a commutative operation; that is, $A \cup B = B \cup A$. Venn diagrams are used to illustrate these operations pictorially.



The intersection of two sets A and B is denoted by $A \cap B$. It results in a set containing the elements that A and B have in common and is defined by:

⁶We discuss permutations and combinations in Chap. 5.

$$A \cap B = \{r \mid r \in A \text{ and } r \in B\}.$$

Suppose $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, then $A \cap B = \{2, 3\}$. Set intersection is a commutative operation; that is, $A \cap B = B \cap A$.

Union and intersection are binary operations but may be extended to more generalized union and intersection operations. For example:

$\cup_{i=1}^n A_i$ denotes the union of n sets.

$\cap_{i=1}^n A_i$ denotes the intersection of n sets.

Set Difference Operations

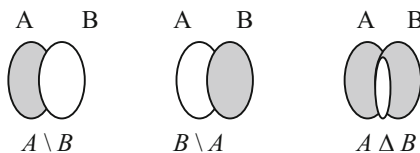
The set difference operation $A \setminus B$ yields the elements in A that are not in B . It is defined by

$$A \setminus B = \{a \mid a \in A \text{ and } a \notin B\}.$$

For A and B defined as $A = \{1, 2\}$ and $B = \{2, 3\}$, we have $A \setminus B = \{1\}$ and $B \setminus A = \{3\}$. Clearly, set difference is not commutative; that is, $A \setminus B \neq B \setminus A$. Clearly, $A \setminus A = \emptyset$ and $A \setminus \emptyset = A$.

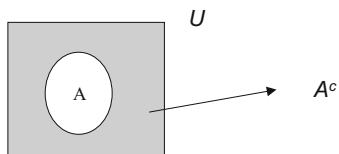
The symmetric difference of two sets A and B is denoted by $A \Delta B$ and is given by: $A \Delta B = A \setminus B \cup B \setminus A$.

The symmetric difference operation is commutative; that is, $A \Delta B = B \Delta A$. Venn diagrams are used to illustrate these operations pictorially.



The complement of a set A (with respect to the universal set U) is the elements in the universal set that are not in A . It is denoted by A^c (or A') and is defined as:

$$A^c = \{u \mid u \in U \text{ and } u \notin A\} = U \setminus A.$$



The complement of the set A is illustrated by the shaded area above.

Table 4.1 Properties of set operations

Property	Description
Commutative	Union and intersection operations are commutative; that is, $S \cup T = T \cup S$ $S \cap T = T \cap S$
Associative	Union and intersection operations are associative; that is, $R \cup (S \cup T) = (R \cup S) \cup T$ $R \cap (S \cap T) = (R \cap S) \cap T$
Identity	The identity under set union is the empty set \emptyset , and the identity under intersection is the universal set U $S \cup \emptyset = \emptyset \cup S = S$ $S \cap U = U \cap S = S$
Distributive	The union operator distributes over the intersection operator and vice versa $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$ $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$
De Morgan's ^a law	The complement of $S \cup T$ is given by: $(S \cup T)^c = S^c \cap T^c$ The complement of $S \cap T$ is given by: $(S \cap T)^c = S^c \cup T^c$

^aDe Morgan's law is named after Augustus De Morgan, a nineteenth-century English mathematician who was a contemporary of George Boole

4.2.2 Properties of Set Theoretical Operations

The set union and set intersection properties are commutative and associative. Their properties are listed in Table 4.1.

These properties may be seen to be true with Venn diagrams, and we give a proof of the distributive property (this proof uses logic which is discussed in Chaps. 5–7).

Proof of Properties (Distributive Property) To show $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

Suppose $x \in A \cap (B \cup C)$, then

$$\begin{aligned}
 x &\in A \wedge x \in (B \cup C) \\
 &\Rightarrow x \in A \wedge (x \in B \vee x \in C) \\
 &\Rightarrow (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C) \\
 &\Rightarrow x \in (A \cap B) \vee x \in (A \cap C) \\
 &\Rightarrow x \in (A \cap B) \cup (A \cap C)
 \end{aligned}$$

Therefore, $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$.

Similarly $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$.

Therefore, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

4.2.3 Russell's Paradox

Bertrand Russell (Fig. 4.1) was a famous British logician, mathematician and philosopher. He was the co-author with Alfred Whitehead of *Principia Mathematica*, which aimed to derive all of the truths of mathematics from logic. Russell's paradox was discovered by Bertrand Russell in 1901, and showed that the system of logicism being proposed by Frege (discussed in Chap. 5) contained a contradiction.

Question (Posed by Russell to Frege)

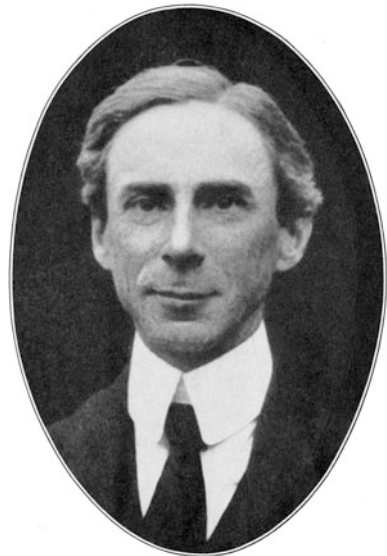
Is the set of all sets that do not contain themselves as members a set?

Russell's Paradox

Let $A = \{S \text{ a set and } S \notin S\}$. Is $A \in A$? Then, $A \in A \Rightarrow A \notin A$ and vice versa. Therefore, a contradiction arises in either case, and there is no such set A .

Two ways of avoiding the paradox were developed in 1908, and these were Russell's theory of types and Zermelo set theory. Russell's theory of types was a response to the paradox by arguing that the set of all sets is ill formed. Russell developed a hierarchy with individual elements the lowest level; sets of elements at

Fig. 4.1 Bertrand Russell



the next level; sets of elements at the next level; and so on. It is then prohibited for a set to contain members of different types.

A set of elements has a different type from its elements, and one cannot speak of the set of all sets that do not contain themselves as members as these are of different types. The other way of avoiding the paradox was Zermelo's axiomatization of set theory (the eight Zermelo-Franklin (ZF) axioms are the most common axiomatization of set theory).

Remark Russell's paradox may also be illustrated by the story of a town that has exactly one barber who is male. *The barber shaves all and only those men in town who do not shave themselves.* The question is who shaves the barber.

If the barber does not shave himself, then according to the rule he is shaved by the barber (i.e. himself). If he shaves himself, then according to the rule he is not shaved by the barber (i.e. himself).

The paradox occurs due to self-reference in the statement, and a logical examination shows that the statement is a contradiction.

4.2.4 Computer Representation of Sets

Sets are fundamental building blocks in mathematics, and so the question arises as to how a set is stored and manipulated in a computer. The representation of a set M on a computer requires a change from the normal view that the order of the elements of the set is irrelevant, and we will need to assume a definite order in the underlying universal set \mathcal{M} from which the set M is defined.

That is, a set is always defined in a computer program with respect to an underlying universal set, and the elements in the universal set are listed in a definite order. Any set M arising in the program that is defined with respect to this universal set \mathcal{M} is a subset of \mathcal{M} . Next, we show how the set M is stored internally on the computer.

The set M is represented in a computer as a string of binary digits $b_1b_2 \dots b_n$ where n is the cardinality of the universal set \mathcal{M} . The bits b_i (where i ranges over the values $1, 2, \dots, n$) are determined according to the rule:

$$\begin{aligned} b_i &= 1 \text{ if } i\text{th element of } \mathcal{M} \text{ is in } M. \\ b_i &= 0 \text{ if } i\text{th element of } \mathcal{M} \text{ is not in } M. \end{aligned}$$

For example, if $\mathcal{M} = \{1, 2, \dots, 10\}$, then the representation of $M = \{1, 2, 5, 8\}$ is given by the bit string 1100100100 where this is given by looking at each element of \mathcal{M} in turn and writing down 1 if it is in M and 0 otherwise.

Similarly, the bit string 0100101100 represents the set $M = \{2, 5, 7, 8\}$, and this is determined by writing down the corresponding element in \mathcal{M} that corresponds to a 1 in the bit string.

Clearly, there is a one-to-one correspondence between the subsets of \mathcal{M} and all possible n -bit strings. Further, the set theoretical operations of set union, intersection and complement can be carried out directly with the bit strings (provided that the sets involved are defined with respect to the same universal set). This involves a bitwise “or” operation for set union; a bitwise “and” operation for set intersection; and a bitwise “not” operation for the set complement operation.

4.3 Relations

A binary relation $R(A, B)$ where A and B are sets is a subset of $A \times B$; that is, $R \subseteq A \times B$. The domain of the relation is A , and the co-domain of the relation is B . The notation aRb signifies that $(a, b) \in R$.

A binary relation $R(A, A)$ is a relation between A and A . This type of relation may always be composed with itself, and its inverse is also a binary relation on A . The identity relation on A is defined by $a i_A a$ for all $a \in A$.

Example 4.4 There are many examples of relations:

- (i) The relation on a set of students in a class where $(a, b) \in R$ if the height of a is greater than the height of b ;
- (ii) The relation between A and B where $A = \{0, 1, 2\}$ and $B = \{3, 4, 5\}$ with R given by:

$$R = \{(0, 3), (0, 4), (1, 4)\}$$

- (iii) The relation less than ($<$) between and \mathbb{R} and \mathbb{R} is given by:

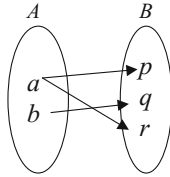
$$\{(x, y) \in \mathbb{R}^2 : x < y\}$$

- (iv) A bank may represent the relationship between the set of accounts and the set of customers by a relation. The implementation of a bank account will often be a positive integer with at most eight decimal digits.

The relationship between accounts and customers may be done with a relation $R \subseteq A \times B$, with the set A chosen to be the set of natural numbers, and the set B chosen to be the set of all human beings alive or dead. The set

A could also be chosen to be $A = \{n \in \mathbb{N} : n < 10^8\}$.

A relation $R(A, B)$ may be represented pictorially. This is referred to as the graph of the relation, and it is illustrated in the diagram below. An arrow from x to y is drawn if (x, y) is in the relation. Thus, for the height relation R given by $\{(a, p), (a, r), (b, q)\}$, an arrow is drawn from a to p , from a to r and from b to q to indicate that (a, p) , (a, r) and (b, q) are in the relation R .



The pictorial representation of the relation makes it easy to see that the height of a is greater than the height of p and r ; and that the height of b is greater than the height of q .

An n -ary relation $R(A_1, A_2, \dots, A_n)$ is a subset of $(A_1 \times A_2 \times \dots \times A_n)$. However, an n -ary relation may also be regarded as a binary relation $R(A, B)$ with $A = A_1 \times A_2 \times \dots \times A_{n-1}$ and $B = A_n$.

4.3.1 Reflexive, Symmetric and Transitive Relations

There are various types of relations including reflexive, symmetric and transitive relations.

- (i) A relation on a set A is *reflexive* if $(a, a) \in R$ for all $a \in A$.
- (ii) A relation R is *symmetric* if whenever $(a, b) \in R$, then $(b, a) \in R$.
- (iii) A relation is *transitive* if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$.

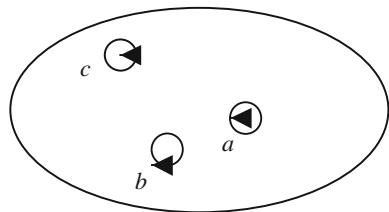
A relation that is reflexive, symmetric and transitive is termed an *equivalence relation*.

Example 4.5 (Reflexive Relation) A relation is reflexive if each element possesses an edge looping around on itself. The relation in Fig. 4.2 is reflexive.

Example 4.6 (Symmetric Relation) The graph of a symmetric relation will show for every arrow from a to b an opposite arrow from b to a . The relation in Fig. 4.3 is symmetric; that is, whenever $(a, b) \in R$, then $(b, a) \in R$.

Example 4.7 (Transitive Relation) The graph of a transitive relation will show that whenever there is an arrow from a to b and an arrow from b to c that there is an

Fig. 4.2 Reflexive relation



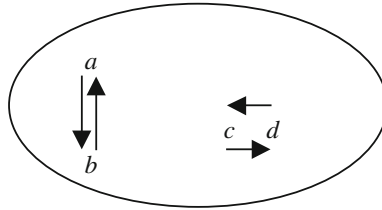
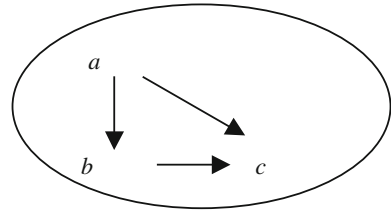


Fig. 4.3 Symmetric relation

Fig. 4.4 Transitive relation



arrow from a to c . The relation in Fig. 4.4 is transitive; that is, whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$.

Example 4.8 (Equivalence Relation) The relation on the set of integers \mathbb{Z} defined by $(a, b) \in R$ if $a - b = 2k$ for some $k \in \mathbb{Z}$ is an equivalence relation, and it partitions the set of integers into two equivalence classes, i.e. the even and odd integers.

Domain and Range of Relation

The domain of a relation $R (A, B)$ is given by $\{a \in A \mid \exists b \in B \text{ and } (a, b) \in R\}$. It is denoted by **dom** R . The domain of the relation $R = \{(a, p), (a, r), (b, q)\}$ is $\{a, b\}$.

The range of a relation $R (A, B)$ is given by $\{b \in B \mid \exists a \in A \text{ and } (a, b) \in R\}$. It is denoted by **rng** R . The range of the relation $R = \{(a, p), (a, r), (b, q)\}$ is $\{p, q, r\}$.

Inverse of a Relation

Suppose $R \subseteq A \times B$ is a relation between A and B , then the inverse relation $R^{-1} \subseteq B \times A$ is defined as the relation between B and A and is given by:

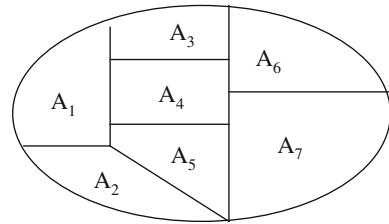
$$b R^{-1} a \text{ if and only if } a R b.$$

That is,

$$R^{-1} = \{(b, a) \in B \times A : (a, b) \in R\}.$$

Example 4.9 Let R be the relation between \mathbb{Z} and \mathbb{Z}^+ defined by $m R n$ if and only if $m^2 = n$. Then, $R = \{(m, n) \in \mathbb{Z} \times \mathbb{Z}^+ : m^2 = n\}$ and $R^{-1} = \{(n, m) \in \mathbb{Z}^+ \times \mathbb{Z} : m^2 = n\}$.

For example, $-3 R 9, -4 R 16, 0 R 0, 16 R^{-1} -4, 9 R^{-1} -3$, etc.

Fig. 4.5 Partitions of A

Partitions and Equivalence Relations

An equivalence relation on A leads to a partition of A , and vice versa for every partition of A there is a corresponding equivalence relation.

Let A be a finite set and let A_1, A_2, \dots, A_n be subsets of A such $A_i \neq \emptyset$ for all i , $A_i \cap A_j = \emptyset$ if $i \neq j$ and $A = \bigcup_i^n A_i = A_1 \cup A_2 \cup \dots \cup A_n$.

The sets A_i partition the set A , and these sets are called the classes of the partition (Fig. 4.5).

Theorem 4.2 (Equivalence Relation and Partitions)

An equivalence relation on A gives rise to a partition of A where the equivalence classes are given by $\text{Class}(a) = \{x \mid x \in A \text{ and } (a, x) \in R\}$. Similarly, a partition gives rise to an equivalence relation R , where $(a, b) \in R$ if and only if a and b are in the same partition.

Proof Clearly, $a \in \text{Class}(a)$ since R is reflexive and clearly the union of the equivalence classes is A . Next, we show that two equivalence classes are either equal or disjoint.

Suppose $\text{Class}(a) \cap \text{Class}(b) \neq \emptyset$. Let $x \in \text{Class}(a) \cap \text{Class}(b)$ and so (a, x) and $(b, x) \in R$. By the symmetric property $(x, b) \in R$ and since R is transitive from (a, x) and (x, b) in R , we deduce that $(a, b) \in R$. Therefore, $b \in \text{Class}(a)$. Suppose y is an arbitrary member of $\text{Class}(b)$, then $(b, y) \in R$; therefore, from (a, b) and (b, y) in R , we deduce that (a, y) is in R . Therefore, since y was an arbitrary member of $\text{Class}(b)$, we deduce that $\text{Class}(b) \subseteq \text{Class}(a)$. Similarly, $\text{Class}(a) \subseteq \text{Class}(b)$ and so $\text{Class}(a) = \text{Class}(b)$.

This proves the first part of the theorem, and for the second part we define a relation R such that $(a, b) \in R$ if a and b are in the same partition. It is clear that this is an equivalence relation.

4.3.2 Composition of Relations

The composition of two relations $R_1(A, B)$ and $R_2(B, C)$ is given by $R_2 \circ R_1$ where $(a, c) \in R_2 \circ R_1$ if and only there exists $b \in B$ such that $(a, b) \in R_1$ and $(b, c) \in R_2$. The composition of relations is associative; that is,

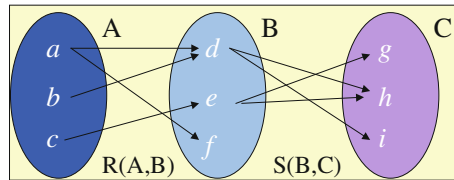


Fig. 4.6 Composition of relations

$$(R_3 \circ R_2) \circ R_1 = R_3 \circ (R_2 \circ R_1).$$

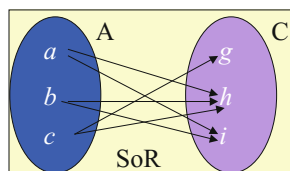
Example 4.10 (Composition of Relations) Consider a library that maintains two files. The first file maintains the serial number s of each book as well as the details of the author a of the book. This may be represented by the relation $R_1 = sR_1a$. The second file maintains the library card number c of its borrowers and the serial number s of any books that they have borrowed. This may be represented by the relation $R_2 = cR_2s$.

The library wishes to issue a reminder to its borrowers of the authors of all books currently on loan to them. This may be determined by the composition of $R_1 \circ R_2$, i.e. $cR_1 \circ R_2a$ if there is book with serial number s such that cR_2s and sR_1a .

Example 4.11 (Composition of Relations) Consider sets $A = \{a, b, c\}$, $B = \{d, e, f\}$, $C = \{g, h, i\}$ and relations $R(A, B) = \{(a, d), (a, f), (b, d), (c, e)\}$ and $S(B, C) = \{(d, h), (d, i), (e, g), (e, h)\}$. Then, we graph these relations and show how to determine the composition pictorially.

$S \circ R$ is determined by choosing $x \in A$ and $y \in C$ and checking if there is a route from x to y in the graph (Fig. 4.6). If so, we join x to y in $S \circ R$. For example, if we consider a and h we see that there is a path from a to d and from d to h and therefore (a, h) is in the composition of S and R .

The union of two relations $R_1(A, B)$ and $R_2(A, B)$ is meaningful (as these are both subsets of $A \times B$). The union $R_1 \cup R_2$ is defined as $(a, b) \in R_1 \cup R_2$ if and only if $(a, b) \in R_1$ or $(a, b) \in R_2$.



Similarly, the intersection of R_1 and R_2 ($R_1 \cap R_2$) is meaningful and is defined as $(a, b) \in R_1 \cap R_2$ if and only if $(a, b) \in R_1$ and $(a, b) \in R_2$. The relation R_1 is a subset of R_2 ($R_1 \subseteq R_2$) if whenever $(a, b) \in R_1$, then $(a, b) \in R_2$.

The inverse of the relation R was discussed earlier and is given by the relation R^{-1} where $R^{-1} = \{(b, a) \mid (a, b) \in R\}$.

The composition of R and R^{-1} yields: $R^{-1} \circ R = \{(a, a) \mid a \in \text{dom } R\} = i_A$ and $R \circ R^{-1} = \{(b, b) \mid b \in \text{dom } R^{-1}\} = i_B$.

4.3.3 Binary Relations

A binary relation R on A is a relation between A and A , and a binary relation can always be composed with itself. Its inverse is a binary relation on the same set. The following are all relations on A :

$$\begin{aligned} R^2 &= R \circ R \\ R^3 &= (R \circ R) \circ R \\ R^0 &= i_A \text{ (identity relation)} \\ R^{-2} &= R^{-1} \circ R^{-1}. \end{aligned}$$

Example 4.12 Let R be the binary relation on the set of all people P such that $(a, b) \in R$ if a is a parent of b . Then, the relation R^n is interpreted as:

- R is the parent relationship.
- R^2 is the grandparent relationship.
- R^3 is the great-grandparent relationship.
- R^{-1} is the child relationship.
- R^{-2} is the grandchild relationship.
- R^{-3} is the great-grandchild relationship.

This can be generalized to a relation R^n on A where $R^n = R \circ R \circ \dots \circ R$ (n times). The transitive closure of the relation R on A is given by:

$$R^* = \bigcup_{i=0}^{\infty} R^i = R^0 \cup R^1 \cup R^2 \cup \dots \cup R^n \cup \dots$$

where R^0 is the reflexive relation containing only each element in the domain of R ; that is, $R^0 = i_A = \{(a, a) \mid a \in \text{dom } R\}$.

The positive transitive closure is similar to the transitive closure except that it does not contain R^0 . It is given by:

$$R^+ = \bigcup_{i=1}^{\infty} R^i = R^1 \cup R^2 \cup \dots \cup R^n \cup \dots$$

$a R^+ b$ if and only if $a R^n b$ for some $n > 0$; that is, there exists $c_1, c_2 \dots c_n \in A$ such that

$$a R c_1, c_1 R c_2, \dots, c_n R b.$$

Parnas⁷ introduced the concept of the limited domain relation (LD relation), and a LD relation L consists of an ordered pair (R_L, C_L) where R_L is a relation and C_L is a subset of $\text{dom } R_L$. The relation R_L is on a set U , and C_L is termed the competence set of the LD relation L . A description of LD relations and a discussion of their properties are in Chap. 2 of [Par:01].

The importance of LD relations is that they may be used to describe program execution. The relation component of the LD relation L describes a set of states such that if execution starts in state x it may terminate in state y . The set U is the set of states. The competence set of L is such that if execution starts in a state that is in the competence set C_L , then it is guaranteed to terminate.

4.3.4 Applications of Relations

A Relational Database Management System (RDBMS) is a system that manages data using the relational model, and examples of such systems include RDMS developed at MIT in the 1970s; Ingres developed at the University of California, Berkeley in the mid-1970s; Oracle developed in the late 1970s; DB2; Informix; and Microsoft SQL Server.

A relation is defined as a set of tuples and is usually represented by a table. A table is data organized in rows and columns, with the data in each column of the table of the same data type. Constraints may be employed to provide restrictions on the kinds of data that may be stored in the relations. Constraints are Boolean expressions which indicate whether the constraint holds or not, and are a way of implementing business rules in the database.

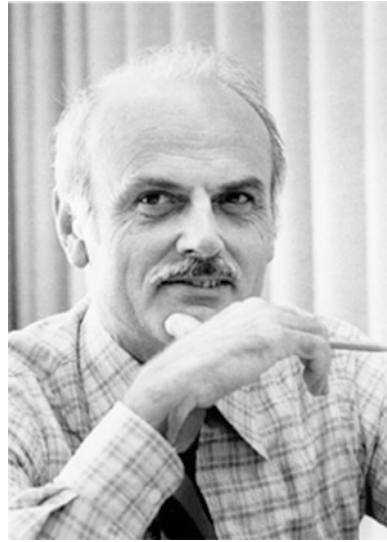
Relations have one or more keys associated with them, and the *key uniquely identifies the row of the table*. An index is a way of providing fast access to the data in a relational database, as it allows the tuple in a relation to be looked up directly (using the index) rather than checking all of the tuples in the relation.

The Structured Query Language (SQL) is a computer language that tells the relational database what to retrieve and how to display it. A stored procedure is executable code that is associated with the database, and it is used to perform common operations on the database.

The concept of a relational database was first described in a paper “*A Relational Model of Data for Large Shared Data Banks*” by Codd [Cod:70]. A relational database is a database that conforms to the relational model, and it may be defined as a set of relations (or tables).

Codd (Fig. 4.7) developed the *relational database model* in the late 1960s, and this is the standard way that information is organized and retrieved from computers. Relational databases are at the heart of systems from hospitals’ patient records to airline flight and schedule information.

⁷Parnas made important contributions to software engineering in the 1970s. He invented information hiding which is used in object-oriented design.

Fig. 4.7 Edgar Codd

A binary relation $R(A, B)$ where A and B are sets is a subset of the Cartesian product $(A \times B)$ of A and B . The domain of the relation is A , and the co-domain of the relation is B . The notation aRb signifies that there is a relation between a and b and that $(a, b) \in R$. An n -ary relation $R(A_1, A_2, \dots, A_n)$ is a subset of the Cartesian product of the n sets. However, an n -ary may also be regarded as a binary relation $R(A, B)$ with $A = A_1 \times A_2 \times \dots \times A_{n-1}$ and $B = A_n$.

The data in the relational model is represented as a mathematical n -ary relation. In other words, a relation is defined as a set of n -tuples, and is usually represented by a table which is a visual representation of the relation, with the data organized in rows and columns.

The basic relational building block is the domain or data type (often called just type). Each row of the table represents one n -tuple (one tuple) of the relation, and the number of tuples in the relation is its cardinality. Consider the PART relation taken from [Dat:81], where this relation consists of a heading and the body. There are five data types representing part numbers, part names, part colours, part weights and locations in which the parts are stored. The body consists of a set of n -tuples, and the PART relation given in Fig. 4.8 is of cardinality six.

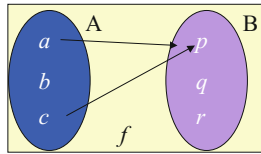
For more information on the relational model and databases, see [ORg:16a].

P#	PName	Colour	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

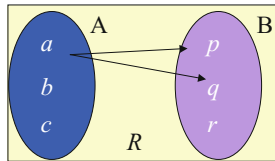
Fig. 4.8 PART relation

4.4 Functions

A function $f: A \rightarrow B$ is a special relation such that for each element $a \in A$ there is exactly (or at most)⁸ one element $b \in B$. This is written as $f(a) = b$.



A function is a relation but not every relation is a function. For example, the relation in the diagram below is not a function since there are two arrows from the element $a \in A$.

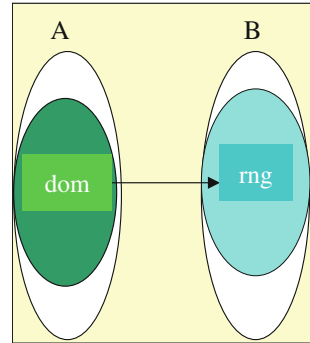


The domain of the function (denoted by **dom** f) is the set of values in A for which the function is defined. The domain of the function is A provided that f is a total function. The co-domain of the function is B . The range of the function (denoted **rng** f) is a subset of the co-domain and consists of:

$$\text{rng } f = \{r \mid r \in B \text{ such that } f(a) = r \text{ for some } a \in A\}.$$

⁸We distinguish between total and partial functions. A total function is defined for all elements in the domain, whereas a partial function may be undefined for one or more elements in the domain.

Fig. 4.9 Domain and range of a partial function



Functions may be partial or total. A *partial function* (or partial mapping) may be undefined for some values of A , and partial functions arise regularly in the computing field (Fig. 4.9). *Total functions* are defined for every value in A , and many functions encountered in mathematics are total.

Example 4.13 (Functions) Functions are an essential part of mathematics and computer science, and there are many well-known functions such as the trigonometric functions $\sin(x)$, $\cos(x)$, and $\tan(x)$; the logarithmic function $\ln(x)$; the exponential functions e^x ; and polynomial functions.

- (i) Consider the partial function $f: \mathbb{R} \rightarrow \mathbb{R}$ where

$$f(x) = \frac{1}{x} \quad (\text{where } x \neq 0).$$

This partial function is defined everywhere except for $x = 0$.

- (ii) Consider the function $f: \mathbb{R} \rightarrow \mathbb{R}$ where

$$f(x) = x^2.$$

Then, this function is defined for all $x \in \mathbb{R}$.

Partial functions often arise in computing as a program may be undefined or fail to terminate for several values of its arguments (e.g. infinite loops). Care is required to ensure that the partial function is defined for the argument to which it is to be applied.

Consider a program P that has one natural number as its input and which fails to terminate for some input values. It prints a single real result and halts when it terminates. Then, P can be regarded as a partial mapping from \mathbb{N} to \mathbb{R} .

$$P : \mathbb{N} \rightarrow \mathbb{R}.$$

Example 4.14 How many total functions $f: A \rightarrow B$ are there from A to B (where A and B are finite sets)?

Each element of A maps to any element of B ; that is, there are $|B|$ choices for each element $a \in A$. Since there are $|A|$ elements in A , the number of total functions is given by:

$$\begin{aligned} & |B| |B| \dots |B| \quad (|A| \text{ times}) \\ & = |B|^{|A|} \quad \text{total functions between } A \text{ and } B. \end{aligned}$$

Example 4.15 How many partial functions $f: A \rightarrow B$ are there from A to B (where A and B are finite sets)?

Each element of A may map to any element of B or to no element of B (as it may be undefined for that element of A). In other words, there are $|B| + 1$ choices for each element of A . As there are $|A|$ elements in A , the number of distinct partial functions between A and B is given by:

$$\begin{aligned} & (|B| + 1) (|B| + 1) \dots (|B| + 1) \quad (|A| \text{ times}) \\ & = (|B| + 1)^{|A|}. \end{aligned}$$

Two partial functions f and g are equal if:

1. $\text{dom } f = \text{dom } g$;
2. $f(a) = g(a)$ for all $a \in \text{dom } f$.

A function f is less defined than a function g ($f \subseteq g$) if the domain of f is a subset of the domain of g , and the functions agree for every value on the domain:

1. $\text{dom } f \subseteq \text{dom } g$;
2. $f(a) = g(a)$ for all $a \in \text{dom } f$.

The composition of functions is similar to the composition of relations. Suppose $f: A \rightarrow B$ and $g: B \rightarrow C$, then $g \circ f: A \rightarrow C$ is a function, and this is written as $g \circ f(x)$ or $g(f(x))$ for $x \in A$.

The composition of functions is not commutative, and this can be seen by an example. Consider the function $f: \mathbb{R} \rightarrow \mathbb{R}$ such that $f(x) = x^2$ and the function $g: \mathbb{R} \rightarrow \mathbb{R}$ such that $g(x) = x + 2$. Then,

$$\begin{aligned} g \circ f(x) &= g(x^2) = x^2 + 2. \\ f \circ g(x) &= f(x + 2) = (x + 2)^2 = x^2 + 4x + 4. \end{aligned}$$

Clearly, $g \circ f(x) \neq f \circ g(x)$ and so composition of functions is not commutative. The composition of functions is associative, as the composition of relations is associative and every function is a relation. For $f: A \rightarrow B$, $g: B \rightarrow C$, and $h: C \rightarrow D$, we have:

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

A function $f: A \rightarrow B$ is *injective (one to one)* if

$$f(a_1) = f(a_2) \Rightarrow a_1 = a_2.$$

For example, consider the function $f: \mathbb{R} \rightarrow \mathbb{R}$ with $f(x) = x^2$. Then, $f(3) = f(-3) = 9$ and so this function is not one to one.

A function $f: A \rightarrow B$ is *surjective (onto)* if given any $b \in B$ there exists an $a \in A$ such that $f(a) = b$ (Fig. 4.10). Consider the function $f: \mathbb{R} \rightarrow \mathbb{R}$ with $f(x) = x + 1$. Clearly, given any $r \in \mathbb{R}$, then $f(r - 1) = r$ and so f is onto.

A function is *bijective* if it is one to one and onto (Fig. 4.11). That is, there is a one-to-one correspondence between the elements in A and B ; for each $b \in B$, there is a unique $a \in A$ such that $f(a) = b$.

The inverse of a relation was discussed earlier, and the relational inverse of a function $f: A \rightarrow B$ clearly exists. The relational inverse of the function may or may not be a function.

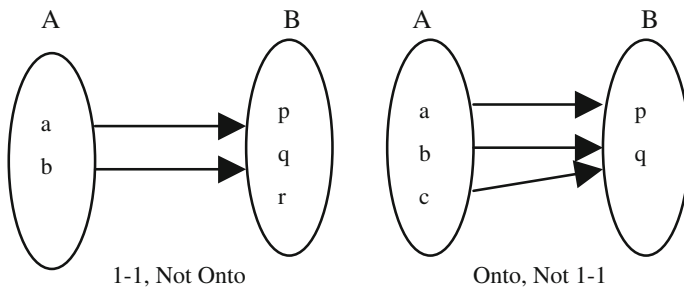


Fig. 4.10 Injective and surjective functions

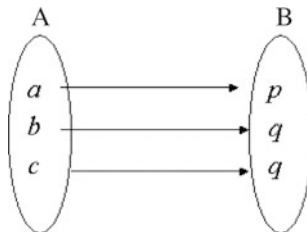


Fig. 4.11 Bijective function (one to one and onto)

However, if the relational inverse is a function, it is denoted by $f^{-1} : B \rightarrow A$. A total function has an inverse (that is a total function) if and only if it is bijective, whereas a partial function has an inverse if and only if it is injective.

The identity function $1_A : A \rightarrow A$ is a function such that $1_A(a) = a$ for all $a \in A$. Clearly, when the inverse of the function exists, then we have that $f^{-1} \circ f = 1_A$ and $f \circ f^{-1} = 1_B$.

Theorem 4.3 (Inverse of Function)

A total function has an inverse if and only if it is bijective.

Proof Suppose $f : A \rightarrow B$ has an inverse f^{-1} , then we show that f is bijective.

We first show that f is one to one.

Suppose $f(x_1) = f(x_2)$, then

$$\begin{aligned} f^{-1}(f(x_1)) &= f^{-1}(f(x_2)) \\ \Rightarrow f^{-1} \circ f(x_1) &= f^{-1} \circ f(x_2) \\ \Rightarrow 1_A(x_1) &= 1_A(x_2) \\ \Rightarrow x_1 &= x_2. \end{aligned}$$

Next, we first show that f is onto. Let

$b \in B$ and let $a = f^{-1}(b)$, then

$$f(a) = f(f^{-1}(b)) = b \text{ and so } f \text{ is surjective.}$$

The second part of the proof is concerned with showing that if $f : A \rightarrow B$ is bijective, then it has an inverse f^{-1} . Clearly, since f is bijective, we have that for each $a \in A$ there exists a unique $b \in B$ such that $f(a) = b$.

Define $g : B \rightarrow A$ by letting $g(b)$ be the unique a in A such that $f(a) = b$. Then, we have that:

$$g \circ f(a) = g(b) = a \text{ and } f \circ g(b) = f(a) = b.$$

Therefore, g is the inverse of f .

4.5 Application of Functions

In this section, we discuss the applications of functions to functional programming, which is quite distinct from the imperative programming languages. Functional programming involves the evaluation of mathematical functions, whereas imperative programming involves the execution of sequential (or iterative) commands that change the state. For example, the assignment statement alters the value of a variable, and so the value of a given variable x may change during program execution.

There are no changes of state for functional programs, and the fact that the value of x will always be the same makes it easier to reason about functional programs than imperative programs. Functional programming languages provide *referential transparency*; that is, equals may be substituted for equals, and if two expressions have equal values, then one can be substituted for the other in any larger expression without affecting the result of the computation.

Functional programming languages use higher-order functions,⁹ recursion, lazy and eager evaluation, monads¹⁰ and Hindley-Milner type inference systems.¹¹ These languages are mainly been used in academia, but there has been some industrial use, including the use of Erlang for concurrent applications in industry. Alonzo Church developed lambda calculus in the 1930s, and it provides an abstract framework for describing mathematical functions and their evaluation. It provides the foundation for functional programming languages. Church employed lambda calculus to prove that there is no solution to the decision problem for first-order arithmetic in 1936 (see Chap. 13 of [ORg:16b]).

Lambda calculus uses transformation rules, and one of these rules is variable substitution. The original calculus developed by Church was untyped, but typed lambda calculi have since been developed. Any computable function can be expressed and evaluated using lambda calculus, but there is no general algorithm to determine whether two arbitrary lambda calculus expressions are equivalent. Lambda calculus influenced functional programming languages such as Lisp, ML and Haskell.

Functional programming uses the notion of higher-order functions. Higher-order takes other functions as arguments and may return functions as results. The derivative function $d/dx.f(x) = f'(x)$ is a higher-order function. It takes a function as an argument and returns a function as a result. For example, the derivative of the function $\sin(x)$ is given by $\cos(x)$. Higher-order functions allow currying which is a technique developed by Schönfinkel. It allows a function with several arguments to be applied to each of its arguments one at a time, with each application returning a new (higher-order) function that accepts the next argument. This allows a function of n -arguments to be treated as n applications of a function with 1-argument.

John McCarthy developed Lisp at MIT in the late 1950s, and this language includes many of the features found in modern functional programming languages.¹² Scheme built upon the ideas in Lisp and was developed at MIT in the

⁹Higher-order functions are functions that take functions as arguments or return a function as a result. They are known as operators (or functionals) in mathematics, and one example is the derivative function d/dx that takes a function as an argument and returns a function as a result.

¹⁰Monads are used in functional programming to express input and output operations without introducing side effects. The Haskell functional programming language makes use of this feature.

¹¹This is the most common algorithm used to perform type inference. Type inference is concerned with determining the type of the value derived from the eventual evaluation of an expression.

¹²Lisp is a multi-paradigm language rather than a functional programming language.

early 1970s. Kenneth Iverson developed APL¹³ in the early 1960s, and this language influenced Backus's FP programming language. Robin Milner designed the ML programming language in the early 1970s. David Turner developed Miranda in the mid-1980s. The Haskell programming language was released in the late 1980s.

4.5.1 Miranda Functional Programming Language

Miranda was developed by David Turner at the University of Kent in the mid-1980s [Tur:85]. It is a non-strict functional programming language; that is, the arguments to a function are not evaluated until they are actually required within the function being called. This is also known as lazy evaluation, and one of its main advantages is that it allows an infinite data structures to be passed as an argument to a function. Miranda is a pure functional language in that there are no side-effect features in the language. The language has been used for:

- Rapid prototyping;
- Specification language;
- Teaching language.

A Miranda program is a collection of equations that define various functions and data structures. It is a strongly typed language with a terse notation.

$$\begin{aligned} z &= \text{sqr } p / \text{sqr } q \\ \text{sqr } k &= k * k \\ p &= a + b \\ q &= a - b \\ a &= 10 \\ b &= 5. \end{aligned}$$

The scope of a formal parameter (e.g. the parameter k above in the function sqr) is limited to the definition of the function in which it occurs.

One of the most common data structures used in Miranda is the list. The empty list is denoted by $[]$, and an example of a list of integers is given by $[1, 3, 4, 8]$. Lists may be appended to by using the “++” operator. For example:

$$[1, 3, 5] ++ [2, 4] = [1, 3, 5, 2, 4].$$

The length of a list is given by the “#” operator:

$$\#[1, 3] = 2.$$

¹³Iverson received the Turing Award in 1979 for his contributions to programming language and mathematical notation. The title of his Turing Award paper was “Notation as a tool of thought”.

The infix operator “:” is employed to prefix an element to the front of a list. For example:

$$5 : [2, 4, 6] \text{ is equal to } [5, 2, 4, 6].$$

The subscript operator “!” is employed for subscripting. For example:

$$\text{Nums} = [5, 2, 4, 6] \quad \text{then} \quad \text{Nums}!0 \text{ is } 5.$$

The elements of a list are required to be of the same type. A sequence of elements that contains mixed types is called a tuple. A tuple is written as follows:

$$\text{Employee} = (\text{“Holmes”}, \text{“221B Baker St. London”}, 50, \text{“Detective”}).$$

A tuple is similar to a record in Pascal, whereas lists are similar to arrays. Tuples cannot be subscripted but their elements may be extracted by pattern matching. Pattern matching is illustrated by the well-known example of the factorial function:

$$\begin{aligned} \text{fac } 0 &= 1 \\ \text{fac } (n + 1) &= (n + 1) * \text{fac } n. \end{aligned}$$

The definition of the factorial function uses two equations, distinguished by the use of different patterns in the formal parameters. Another example of pattern matching is the reverse function on lists:

$$\begin{aligned} \text{reverse } [] &= [] \\ \text{reverse } (a : x) &= \text{reverse } x ++ [a]. \end{aligned}$$

Miranda is a higher-order language, and it allows functions to be passed as parameters and returned as results. Currying is allowed, and this allows a function of n -arguments to be treated as n applications of a function with 1-argument. Function application is left associative; that is, $f \ x \ y$ means $(f \ x) \ y$. That is, the result of applying the function f to x is a function, and this function is then applied to y . Every function with two or more arguments in Miranda is a higher-order function.

4.6 Review Questions

1. What is a set? A relation? A function?
2. Explain the difference between a partial and a total function.
3. Explain the difference between a relation and a function.

4. Determine $A \times B$ where $A = \{a, b, c, d\}$ and $B = \{1, 2, 3\}$.
5. Determine the symmetric difference $A \Delta B$ where $A = \{a, b, c, d\}$ and $B = \{c, d, e\}$.
6. What is the graph of the relation \leq on the set $A = \{2, 3, 4\}$?
7. What is the composition of S and R (i.e. $S \circ R$), where R is a relation between A and B , and S is a relation between B and C ? The sets A, B, C are defined as $A = \{a, b, c, d\}$, $B = \{e, f, g\}$, $C = \{h, i, j, k\}$ and $R = \{(a, e), (b, e), (b, g), (c, e), (d, f)\}$ with $S = \{(e, h), (e, k), (f, j), (f, k), (g, h)\}$.
8. What is the domain and range of the relation R where $R = \{(a, p), (a, r), (b, q)\}$?
9. Determine the inverse relation R^{-1} where $R = \{(a,2), (a,5), (b,3), (b,4), (c,1)\}$.
10. Determine the inverse of the function $f: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ defined by

$$f(x) = \frac{x-2}{x-3} \quad (x \neq 3) \quad \text{and} \quad f(3) = 1.$$

11. Give examples of injective, surjective and bijective functions.
12. Let $n \geq 2$ be a fixed integer. Consider the relation \equiv defined by $\{(p, q) : p, q \in \mathbb{Z}, n \mid (q - p)\}$.
 - (a) Show \equiv is an equivalence relation.
 - (b) What are the equivalence classes of this relation?
13. Describe the differences between imperative programming languages and functional programming languages.

4.7 Summary

This chapter provided an introduction to set theory, relations and functions. Sets are collections of well-defined objects; a relation between A and B indicates relationships between members of the sets A and B ; and functions are a special type of relation where there is at most one relationship for each element $a \in A$ with an element in B .

A set is a collection of well-defined objects that contains no duplicates. There are many examples of sets such as the set of natural numbers \mathbb{N} , the integer numbers \mathbb{Z} and so on.

The Cartesian product allows a new set to be created from existing sets. The Cartesian product of two sets S and T (denoted $S \times T$) is the set of ordered pairs $\{(s, t) \mid s \in S, t \in T\}$.

A binary relation $R(A, B)$ is a subset of the Cartesian product $(A \times B)$ of A and B where A and B are sets. The domain of the relation is A , and the co-domain of the relation is B . The notation aRb signifies that there is a relation between a and b and that $(a, b) \in R$. An n -ary relation $R(A_1, A_2, \dots, A_n)$ is a subset of $(A_1 \times A_2 \times \dots \times A_n)$.

A total function $f: A \rightarrow B$ is a special relation such that for each element $a \in A$ there is exactly one element $b \in B$. This is written as $f(a) = b$. A function is a relation but not every relation is a function.

The domain of the function (denoted by **dom** f) is the set of values in A for which the function is defined. The domain of the function is A provided that f is a total function. The co-domain of the function is B .

Functional programming is quite distinct from imperative programming in that there is no change of state, and the value of the variable x remains the same during program execution. This makes functional programs easier to reason about than imperative programs.

5.1 Introduction

Logic is concerned with reasoning and with establishing the validity of arguments. It allows conclusions to be deduced from premises according to logical rules, and the logical argument establishes the truth of the conclusion provided that the premises are true.

The origins of logic are with the Greeks who were interested in the nature of truth. The sophists (e.g. Protagoras and Gorgias) were teachers of rhetoric, who taught their pupils techniques in winning an argument and convincing an audience. Plato explores the nature of truth in some of his dialogues, and he is critical of the position of the sophists who argue that there is no absolute truth, and that truth instead is always relative to some frame of reference. The classic sophist position is stated by Protagoras “*Man is the measure of all things: of things which are, that they are, and of things which are not, that they are not*”. In other words, what is true for you is true for you, and what is true for me is true for me.

Socrates had a reputation for demolishing an opponent’s position, and the Socratic enquiry consisted of questions and answers in which the opponent would be led to a conclusion incompatible with his original position. The approach was similar to a *reductio ad absurdum* argument, although Socrates was a moral philosopher who did no theoretical work on logic.

Aristotle did important work on logic, and he developed a system of logic, *syllogistic logic*, that remained in use up to the nineteenth century. Syllogistic logic is a “term-logic”, with letters used to stand for the individual terms. A syllogism consists of two premises and a conclusion, where the conclusion is a valid deduction from the two premises. Aristotle also did some early work on modal logic and was the founder of the field.

The Stoics developed an early form of propositional logic, where the assertibles (propositions) have a truth-value such that at any time they are either true or false. The assertibles may be simple or non-simple, and various connectives such as

conjunctions, disjunctions and implication are used in forming more complex assertibles.

George Boole developed his symbolic logic in the mid-1800s, and it later formed the foundation for digital computing. Boole argued that logic should be considered as a separate branch of mathematics, rather than a part of philosophy. He argued that there are mathematical laws to express the operation of reasoning in the human mind, and he showed how Aristotle’s syllogistic logic could be reduced to a set of algebraic equations.

Logic plays a key role in reasoning and deduction in mathematics, but it is considered a separate discipline to mathematics. There were attempts in the early twentieth century to show that all mathematics can be derived from formal logic, and that the formal system of mathematics would be complete, with all the truths of mathematics provable in the system (see Chap. 13 of [ORg:16b]). However, this program failed when the Austrian logician, Kurt Gödel, showed that that there are truths in the formal system of arithmetic that cannot be proved within the system (i.e. first-order arithmetic is incomplete).

5.2 Syllogistic Logic

Early work on logic was done by Aristotle in the fourth century B.C. in the *Organon* [Ack:94]. Aristotle regarded logic as a useful tool of enquiry into any subject, and he developed *syllogistic logic*. This is a form of reasoning in which a conclusion is drawn from two premises, where each premise is in a subject–predicate form. A common or middle term is present in each of the two premises but not in the conclusion. For example:

All Greeks are mortal
Socrates is a Greek

Therefore Socrates is mortal

The common (or middle) term in this example is “Greek”. It occurs in both premises but not in the conclusion. The above argument is valid, and Aristotle studied and classified the various types of syllogistic arguments to determine those that were valid or invalid. Each premise contains a subject and a predicate, and the middle term may act as subject or a predicate. Each premise is a positive or negative affirmation, and an affirmation may be universal or particular. The universal and particular affirmations and negatives are described in Table 5.1.

This leads to four basic forms of syllogistic arguments (Table 5.2) where the middle is the subject of both premises; the predicate of both premises; and the subject of one premise and the predicate of the other premise.

Table 5.1 Types of syllogistic premises

Type	Symbol	Example
Universal affirmative	G A M	All Greeks are mortal
Universal negative	G E M	No Greek is mortal
Particular affirmative	G I M	Some Greek is mortal
Particular negative	G O M	Some Greek is not mortal

Table 5.2 Forms of syllogistic premises

	Form (i)	Form (ii)	Form (iii)	Form (iv)
Premise 1	M P	P M	P M	M P
Premise 2	M S	S M	M S	S M
Conclusion	S P	S P	S P	S P

There are four types of premises (A, E, I, O) and therefore sixteen sets of premise pairs for each of the forms above. However, only some of these premise pairs will yield a valid conclusion. Aristotle went through every possible premise pair to determine whether a valid argument may be derived. The syllogistic argument above is of form (iv) and is valid:

G A M
 S I G

 S I M

Syllogistic logic is a “term-logic” with letters used to stand for the individual terms. Syllogistic logic was the first attempt at a science of logic, and it remained in use up to the nineteenth century. There are many limitations to what it may express, and on its suitability as a representation of how the mind works.

5.3 Paradoxes and Fallacies

A paradox is a statement that apparently contradicts itself, and it presents a situation that appears to defy logic. Some logical paradoxes have a solution, whereas others are contradictions or invalid arguments. There are many examples of paradoxes, and they often arise due to self-reference in which one or more statements refer to each other. We discuss several paradoxes such as the *liar paradox* and the *sorites paradox*, which were invented by Eubulides of Miletus, and the *barber paradox*, which was introduced by Russell to explain the contradictions in naïve set theory.

An example of the *liar paradox* is the statement “Everything that I say is false”, which is made by the liar. This looks like a normal sentence, but it is also saying something about itself as a sentence. If the statement is true, then the statement must be false, since the meaning of the sentence is that every statement (including the current statement) made by the liar is false. If the current statement is false, then the statement that everything that I say is false is false, and so this must be a true statement.

The *Epimenides paradox* is a variant of the liar paradox. Epimenides was a Cretan who allegedly stated “*All Cretans are liars*”. If the statement is true, then since Epimenides is Cretan, he must be a liar, and so the statement is false and we have a contradiction. However, if we assume that the statement is false and that Epimenides is lying about all Cretan being liars, then we may deduce (without contradiction) that there is at least one Cretan who is truthful. So in this case, the paradox can be avoided.

The *sorites paradox* (paradox of the heap) involves a heap of sand in which grains are individually removed. It is assumed that removing a single grain of sand does not turn a heap into a non-heap, and the paradox is to consider what happens after when the process is repeated often enough. Is a single remaining grain a heap? When does it change from being a heap to a non-heap? This paradox may be avoided by specifying a fixed boundary of the number of grains of sand required to form a heap, or to define a heap as a collection of multiple grains (≥ 2 grains). Then, any collection of grains of sand less than this boundary is not a heap.

The *barber paradox* is a variant of Russell’s paradox (a contradiction in naïve set theory), which was discussed in Chap. 4. In a village, there is a barber who shaves everyone who does not shave himself, and no one else. Who shaves the barber? The answer to this question results in a contradiction, as the barber cannot shave himself, since he shaves only those who do not shave themselves. Further, as the barber does not shave himself then he falls into the group of people who would be shaved by the barber (himself). Therefore, we conclude that there is no such barber.

The purpose of a debate is to convince an audience of the correctness of your position and to challenge and undermine your opponent’s position. Often, the arguments made are factual, but occasionally individuals skilled in rhetoric and persuasion will introduce bad arguments as a way to persuade the audience. Aristotle studied and classified bad arguments (known as *fallacies*), and these include fallacies such as the *ad hominem* argument; the *appeal to authority* argument; and the *straw man* argument. The fallacies are described in more detail in Table 5.3.

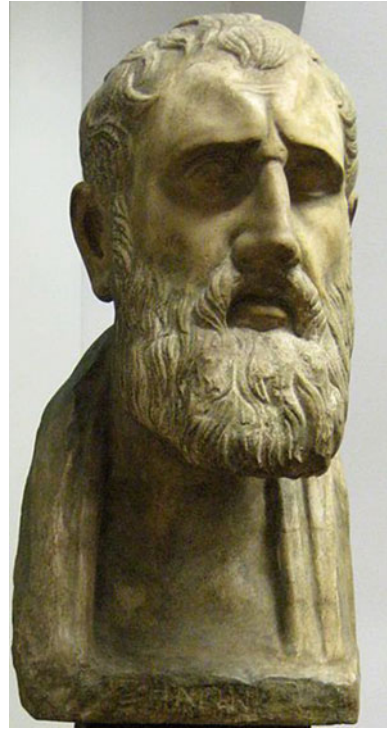
Table 5.3 Fallacies in arguments

Fallacy	Description/Example
Hasty/Accident generalization	This is a bad argument that involves a generalization that disregards exceptions
Slippery slope	This argument outlines a chain reaction leading to a highly undesirable situation that will occur if a certain situation is allowed. The claim is that even if one step is taken onto the slippery slope, then we will fall all the way down to the bottom
Against the person <i>Ad Hominem</i>	The focus of this argument is to attack the person rather than the argument that the person has made
Appeal to people <i>Ad Populum</i>	This argument involves an appeal to popular belief to support an argument, with a claim that the majority of the population supports this argument. However, popular opinion is not always correct
Appeal to authority (<i>Ad Verecundiam</i>)	This argument is when an appeal is made to an authoritative figure to support an argument, and where the authority is not an expert in this area
Appeal to pity (<i>Ad Misericordiam</i>)	This is where the arguer tries to get people to accept a conclusion by making them feel sorry for someone
Appeal to ignorance	The arguer makes the case that there is no conclusive evidence on the issue at hand and that therefore his conclusion should be accepted
Straw man argument	The arguer sets up a version of an opponent's position of his argument and defeats this watered down version of his opponent's position
Begging the question	This is a circular argument where the arguer relies on a premise that says the same thing as the conclusion and without providing any real evidence for the conclusion
Red herring	The arguer goes off on a tangent that has nothing to do with the argument in question
False dichotomy	The arguer presents the case that there are only two possible outcomes (often there are more). One of the possible outcomes is then eliminated leading to the desired outcome. The argument suggests that there is only one outcome

5.4 Stoic Logic

The Stoic school¹ was founded in the Hellenistic period by Zeno of Citium (in Cyprus) in the late fourth/early third century B.C. (Fig. 5.1). The school presented its philosophy as a way of life, and it emphasized ethics as the main focus of human knowledge. The Stoics stressed the importance of living a good life in harmony with nature.

¹The origin of the word Stoic is from the *Stoa Poikile* (Στοα Ποικίλη), which was a covered walkway in the Agora of Athens. Zeno taught his philosophy in a public space at this location, and his followers became known as Stoics.

Fig. 5.1 Zeno of Citium

The Stoics recognized the importance of reason and logic, and Chrysippus, the head of the Stoics in the third century B.C., developed an early version of propositional logic. This was a system of deduction in which the smallest unanalyzed expressions are assertibles (Stoic equivalent of propositions). The assertibles have a truth-value such that at any moment of time they are either true or false. True assertibles are viewed as facts in the Stoic system of logic, and false assertibles are defined as the contradictories of true ones.

Truth is temporal, and assertions may change their truth-value over time. The assertibles may be simple or non-simple (more than one assertible), and there may be present tense, past tense and future tense assertibles. Chrysippus distinguished between simple and compound propositions, and he introduced a set of logical connectives for conjunction, disjunction and implication that are used to form non-simple assertibles from existing assertibles.

The conjunction connective is of the form “*both .. and ..*”, and it has two conjuncts. The disjunction connective is of the form “*either .. or .. or ..*”, and it consists of two or more disjuncts. Conditionals are formed from the connective “*if*”, and they consist of an antecedent and a consequence.

His deductive system included various logical argument forms such as *modus ponens* and *modus tollens*. His propositional logic differed from syllogistic logic, in that the Stoic logic was based on propositions (or statements) as distinct from

Aristotle's term-logic. However, he could express the universal affirmation in syllogistic logic (e.g. All As are B) by rephrasing it as a conditional statement that if something is A then it is B.

Chrysippus's propositional logic did not replace Aristotle's syllogistic logic, and syllogistic logic remained in use up to the mid-nineteenth century. George Boole developed his symbolic logic in the mid-1800s, and his logic later formed the foundation for digital computing. Boole's symbolic logic is discussed in the next section.

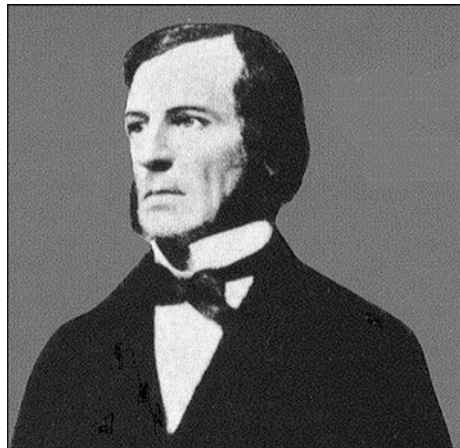
5.5 Boole's Symbolic Logic

George Boole (Fig. 5.2) was born in Lincoln, England, in 1815. His father (a cobbler who was interested in mathematics and optical instruments) taught him mathematics and showed him how to make optical instruments. Boole inherited his father's interest in knowledge, and he was self-taught in mathematics and Greek. He taught at various schools near Lincoln, and he developed his mathematical knowledge by working his way through Newton's *Principia*, as well as applying himself to the work of mathematicians such as Laplace and Lagrange.

He published regular papers from his early twenties, and these included contributions to probability theory, differential equations and finite differences. He developed his symbolic algebra, which is the foundation for modern computing, and he is considered (along with Babbage) to be one of the grandfathers of computing. His work was theoretical, and he never actually built a computer or calculating machine. *However, Boole's symbolic logic was the perfect mathematical model for switching theory, and for the design of digital circuits.*

Boole became interested in formulating a calculus of reasoning, and he published a pamphlet titled "Mathematical Analysis of Logic" in 1847 [Boo:48]. This short book developed novel ideas on a logical method, and he argued that logic should be considered as a separate branch of mathematics, rather than a part of philosophy.

Fig. 5.2 George Boole



He argued that there are mathematical laws to express the operation of reasoning in the human mind, and he showed how Aristotle’s syllogistic logic could be reduced to a set of algebraic equations. He corresponded regularly on logic with Augustus De Morgan.²

He introduced two quantities “0” and “1” with the quantity 1 used to represent the universe of thinkable objects (i.e. the universal set), and the quantity 0 represents the absence of any objects (i.e. the empty set). He then employed symbols such as x, y, z , to represent collections or classes of objects given by the meaning attached to adjectives and nouns. Next, he introduced three operators ($+$, $-$ and \times) that combined classes of objects.

The expression xy (i.e. x multiplied by y or $x \times y$) combines the two classes x, y to form the new class xy (i.e. the class whose objects satisfy the two meanings represented by the classes x and y). Similarly, the expression $x + y$ combines the two classes x, y to form the new class $x + y$ (that satisfies either the meaning represented by class x or class y). The expression $x - y$ combines the two classes x, y to form the new class $x - y$. This represents the class (that satisfies the meaning represented by class x but not class y). The expression $(1 - x)$ represents objects that do not have the attribute that represents class x .

Thus, if $x = \text{black}$ and $y = \text{sheep}$, then xy represents the class of black sheep. Similarly, $(1 - x)$ would represent the class obtained by the operation of selecting all things in the world except black things; $x(1 - y)$ represents the class of all things that are black but not sheep; and $(1 - x)(1 - y)$ would give us all things that are neither sheep nor black.

He showed that these symbols obeyed a rich collection of algebraic laws and could be added, multiplied, etc., in a manner that is similar to real numbers. These symbols may be used to reduce propositions to equations, and algebraic rules may be employed to solve the equations. The rules include:

1.	$x + 0 = x$	(Additive Identity)
2.	$x + (y + z) = (x + y) + z$	(Associative)
3.	$x + y = y + x$	(Commutative)
4.	$x + (1 - x) = 1$	
5.	$x \cdot 1 = x$	(Multiplicative identity)
6.	$x \cdot 0 = 0$	
7.	$x + 1 = 1$	
8.	$xy = yx$	(Commutative)
9.	$x(yz) = (xy)z$	(Associative)
10.	$x(y + z) = xy + xz$	(Distributive)
11.	$x(y - z) = xy - xz$	(Distributive)
12.	$x^2 = x$	(Idempotent)

²De Morgan was a nineteenth British mathematician based at University College London. De Morgan’s laws in Set Theory and Logic state that: $(A \cup B)^c = A^c \cap B^c$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$.

These operations are similar to the modern laws of set theory with the set union operation represented by “+”, and the set intersection operation is represented by multiplication. The universal set is represented by “1” and the empty by “0”. The associative and distributive laws hold. Finally, the set complement operation is given by $(1 - x)$.

Boole applied the symbols to encode Aristotle's syllogistic logic, and he showed how the syllogisms could be reduced to equations. This allowed conclusions to be derived from premises by eliminating the middle term in the syllogism. He refined his ideas on logic further in his book “*An Investigation of the Laws of Thought*” [Boo:58]. This book aimed to identify the fundamental laws underlying reasoning in the human mind and to give expression to these laws in the symbolic language of a calculus.

He considered the equation $x^2 = x$ to be a fundamental laws of thought. It allows the principle of contradiction to be expressed (i.e. for an entity to possess an attribute and at the same time not to possess it):

$$\begin{aligned}x^2 &= x \\ \Rightarrow x - x^2 &= 0 \\ \Rightarrow x(1-x) &= 0\end{aligned}$$

For example, if x represents the class of horses, then $(1 - x)$ represents the class of “not-horses”. The product of two classes represents a class whose members are common to both classes. Hence, $x(1 - x)$ represents the class whose members are at once both horses and “not-horses”, and the equation $x(1 - x) = 0$ expresses that fact that there is no such class. That is, it is the empty set.

Boole contributed to other areas in mathematics including differential equations, finite differences³ and to the development of probability theory. Des McHale has written an interesting biography of Boole [McH:85]. Boole's logic appeared to have no practical use, but this changed with Claude Shannon's 1937 Master's Thesis, which showed its applicability to switching theory and to the design of digital circuits.

5.5.1 Switching Circuits and Boolean Algebra

Claude Shannon showed in his famous Master's Thesis that Boole's symbolic algebra provided the perfect mathematical model for switching theory and for the design of digital circuits. It may be employed to optimize the design of systems of electromechanical relays, and circuits with relays solve Boolean algebra problems. The use of the properties of electrical switches to process logic is the basic concept that underlies all modern electronic digital computers.

³Finite differences are a numerical method used in solving differential equations.

Modern electronic computers use millions (billions) of transistors that act as switches and can change state rapidly. The use of switches to represent binary values is the foundation of modern computing. Digital computers use the binary digits 0 and 1, and a high voltage represents the binary value 1 with a low voltage representing the binary value 0.

A silicon chip may contain billions of tiny electronic switches arranged into logical gates. The basic logic gates are AND, OR and NOT, and these gates may be combined in various ways to perform more complex tasks such as binary arithmetic. Each logic gate has binary value inputs and produces binary value outputs. Boolean logical operations may be implemented by electronic AND, OR and NOT gates, and more complex circuits may be designed from these fundamental building blocks.

The example in Fig. 5.3 is that of an “AND” gate which produces the binary value 1 as output only if both inputs are 1. Otherwise, the result will be the binary value 0. Figure 5.4 is an “OR” gate which produces the binary value 1 as output if any of its inputs is 1. Otherwise, it will produce the binary value 0.

Fig. 5.3 Binary AND operation

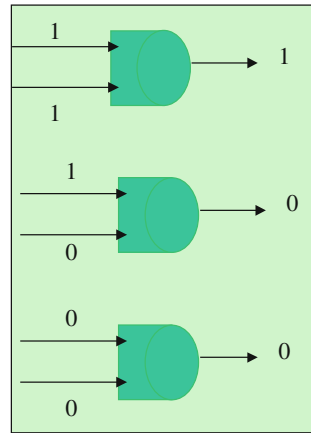
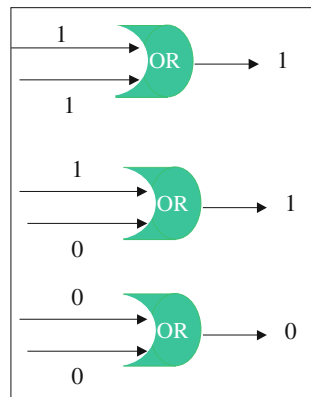


Fig. 5.4 Binary OR operation



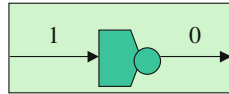
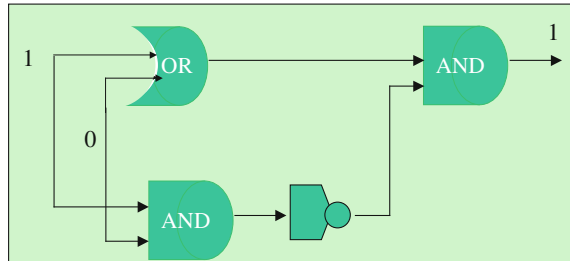


Fig. 5.5 NOT operation

Fig. 5.6 Half-adder



Finally, a NOT gate (Fig. 5.5) accepts only a single input which it reverses. That is, if the input is “1”, then value “0” is produced and vice versa.

The logic gates may be combined to form more complex circuits. The example in Fig. 5.6 is that of a half-adder of $1 + 0$. The inputs to the top OR gate are 1 and 0 which yields the result of 1. The inputs to the bottom AND gate are 1 and 0 which yields the result 0, which is then inverted through the NOT gate to yield binary 1. Finally, the last AND gate receives two 1's as input, and the binary value 1 is the result of the addition.

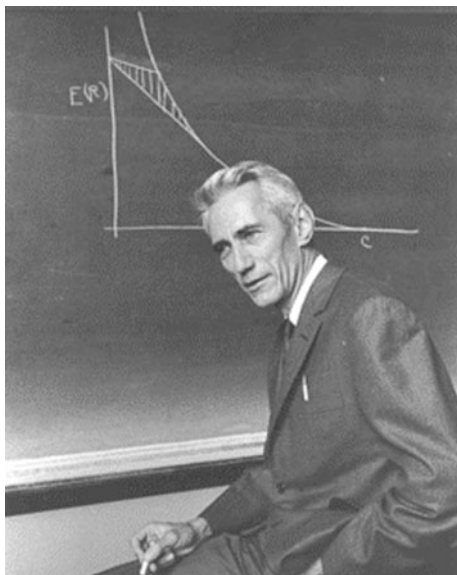
The half-adder computes the addition of two arbitrary binary digits, but it does not calculate the carry. It may be extended to a full-adder that provides a carry for addition.

5.6 Application of Symbolic Logic to Digital Computing

Claude Shannon (Fig. 5.7) was an American mathematician and engineer who made fundamental contributions to computing. He was the first person⁴ to see the applicability of Boolean algebra to simplify the design of circuits and telephone routing switches. He showed that Boole's symbolic logic developed in the nineteenth century provided the perfect mathematical model for switching theory and for the subsequent design of digital circuits and computers.

Vannevar Bush [ORg:13] was Shannon's supervisor at MIT, and Shannon's initial work was to improve Bush's mechanical computing device known as the Differential Analyser. This machine had a complicated control circuit that was

⁴Victor Shestakov at Moscow State University also proposed a theory of electric switches based on Boolean algebra around the same time as Shannon. However, his results were published in Russian in 1941, whereas Shannon's were published in 1937.

Fig. 5.7 Claude Shannon

composed of one hundred switches that could be automatically opened and closed by an electromagnet. Shannon's insight was his realization that an electronic circuit is similar to Boole's symbolic algebra, and he showed how Boolean algebra could be employed to optimize the design of systems of electromechanical relays used in the analog computer. He also realized that circuits with relays could solve Boolean algebra problems.

Shannon's influential *Master's Thesis is a key milestone in computing*, and it shows how to lay out circuits according to Boolean principles. It provides the theoretical foundation of switching circuits, and *his insight of using the properties of electrical switches to do Boolean logic is the basic concept that underlies all electronic digital computers*.

Shannon realized that you could combine switches in circuits in such a manner as to carry out symbolic logic operations. This allowed binary arithmetic and more complex mathematical operations to be performed by relay circuits. He designed a circuit, which could add binary numbers, and he later designed circuits that could make comparisons and thus is capable of performing a conditional statement. *This was the birth of digital logic and the digital computing age*.

He showed in his Master's thesis "A Symbolic Analysis of Relay and Switching Circuits" [Sha:37] that the binary digits (i.e. 0 and 1) can be represented by electrical switches. The implications of true and false being denoted by the binary digits one and zero were enormous, since it allowed binary arithmetic and more complex mathematical operations to be performed by relay circuits. This provided electronics engineers with the mathematical tool they needed to design digital electronic circuits and provided the foundation of digital electronic design.

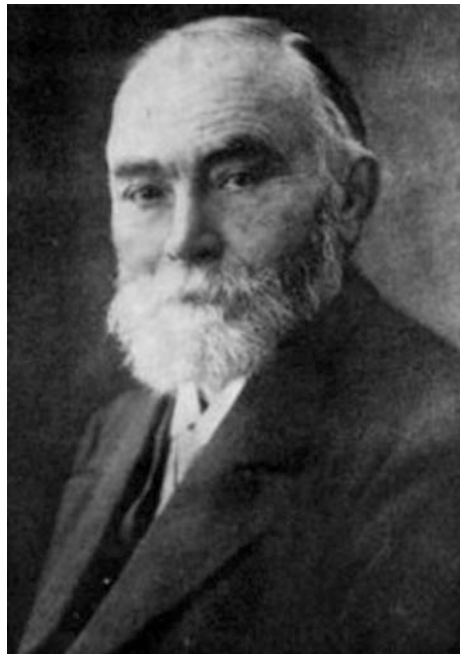
The design of circuits and telephone routing switches could be simplified with Boole's symbolic algebra. Shannon showed how to lay out circuitry according to Boolean principles, and his Master's thesis became the foundation for the practical design of digital circuits. These circuits are fundamental to the operation of modern computers and telecommunication systems, and his insight of using the properties of electrical switches to do Boolean logic is the basic concept that underlies all electronic digital computers.

5.7 Frege

Gottlob Frege (Fig. 5.8) was a German mathematician and logician who is considered (along with Boole) to be one of the founders of modern logic. He also made important contributions to the foundations of mathematics, and he attempted to show that all of the basic truths of mathematics (or at least of arithmetic) could be derived from a limited set of logical axioms (this approach is known as *logicism*).

He invented predicate logic and the universal and existential quantifiers, and predicate logic was a significant advance on Aristotle's syllogistic logic. Predicate logic is described in more detail in Chap. 6.

Fig. 5.8 Gottlob Frege



Frege's first logical system, the 1879 *Begriffsschrift*, contained nine axioms and one rule of inference. It was the axiomatization of logic, and it was complete in its treatment of propositional logic and first-order predicate logic. He published several important books on logic, including *Begriffsschrift*, in 1879; *Die Grundlagen der Arithmetik* (The Foundations of Arithmetic) in 1884; and the two-volume work *Grundgesetze der Arithmetik* (Basic Laws of Arithmetic), which were published in 1893 and 1903. These books described his invention of axiomatic predicate logic; the use of quantified variables; and the application of his logic to the foundations of arithmetic.

Frege presented his predicate logic in his books, and he began to use it to define the natural numbers and their properties. He had intended producing three volumes of the Basic Laws of Arithmetic, with the later volumes dealing with the real numbers and their properties. However, Bertrand Russell discovered a contradiction in Frege's system (see Russell's paradox in Chap. 4), which he communicated to Frege shortly before the publication of the second volume. Frege was astounded by the contradiction and he struggled to find a satisfactory solution, and Russell later introduced the theory of types in the *Principia Mathematica* as a solution.

5.8 Review Questions

1. What is logic?
2. What is a fallacy?
3. Give examples of fallacies in arguments in natural language (e.g. in politics, marketing, debates)
4. Investigate some of the early paradoxes (e.g. the Tortoise and Achilles paradox or the arrow in flight paradox) and give your interpretation of the paradox.
5. What is syllogistic logic and explain its relevance.
6. What is stoic logic and explain its relevance.
7. Explain the significance of the equation $x^2 = x$ in Boole's symbolic logic.
8. Describe how Boole's symbolic logic provided the foundation for digital computing.
9. Describe Frege's contributions to logic.

5.9 Summary

This chapter gave a short introduction to logic, and logic is concerned with reasoning and with establishing the validity of arguments. It allows conclusions to be deduced from premises according to logical rules, and the logical argument establishes the truth of the conclusion provided that the premises are true.

The origins of logic are with the Greeks who were interested in the nature of truth. Socrates had a reputation for demolishing an opponent's position (it meant that he did not win any friends with in debate), and the Socratean enquiry consisted of questions and answers in which the opponent would be led to a conclusion incompatible with his original position. His approach was similar to a *reductio ad absurdum* argument, and its effect was to show that his opponent's position was incoherent and untenable.

Aristotle did important work on logic, and he developed a system of logic, *syllogistic logic*, that remained in use up to the nineteenth century. Syllogistic logic is a "term-logic", with letters used to stand for the individual terms. A syllogism consists of two premises and a conclusion, where the conclusion is a valid deduction from the two premises. The Stoics developed an early form of propositional logic, where the assertibles (propositions) have a truth-value such that at any time they are either true or false.

George Boole developed his symbolic logic in the mid-1800s, and it later formed the foundation for digital computing. Boole argued that logic should be considered as a separate branch of mathematics, rather than a part of philosophy. He argued that there are mathematical laws to express the operation of reasoning in the human mind, and he showed how Aristotle's syllogistic logic could be reduced to a set of algebraic equations.

Gottlob Frege made important contributions to logic and to the foundations of mathematics. He attempted to show that all of the basic truths of mathematics (or at least of arithmetic) could be derived from a limited set of logical axioms (this approach is known as *logicism*). He invented predicate logic and the universal and existential quantifiers, and predicate logic was a significant advance on Aristotle's syllogistic logic

6.1 Introduction

Logic is the study of reasoning and the validity of arguments, and it is concerned with the truth of statements (propositions) and the nature of truth. Formal logic is concerned with the form of arguments and the principles of valid inference. Valid arguments are truth preserving, and for a valid deductive argument the conclusion will always be true if the premises are true.

Propositional logic is the study of propositions, where a proposition is a statement that is either true or false. Propositions may be combined with other propositions (with a logical connective) to form compound propositions. Truth tables are used to give operational definitions of the most important logical connectives, and they provide a mechanism to determine the truth-values of more complicated logical expressions.

Propositional logic may be used to encode simple arguments that are expressed in natural language, and to determine their validity. The validity of an argument may be determined from truth tables, or using the inference rules such as modus ponens to establish the conclusion via deductive steps.

Predicate logic allows complex facts about the world to be represented, and new facts may be determined via deductive reasoning. Predicate calculus includes predicates, variables and quantifiers, and a *predicate* is a characteristic or property that the subject of a statement can have. A predicate may include variables, and statements with variables become propositions once the variables are assigned values.

The universal quantifier is used to express a statement such as that all members of the domain of discourse have property P . This is written as $(\forall x) P(x)$, and it expresses the statement that the property $P(x)$ is true for all x .

The existential quantifier states that there is at least one member of the domain of discourse that has property P . This is written as $(\exists x)P(x)$.

6.2 Propositional Logic

Propositional logic is the study of propositions where a proposition is a statement that is either true or false. There are many examples of propositions such as “ $1 + 1 = 2$ ” which is a true proposition, and the statement that “Today is Wednesday” which is true if today is Wednesday and false otherwise. The statement $x > 0$ is not a proposition as it contains a variable x , and it is only meaningful to consider its truth or falsity only when a value is assigned to x . Once the variable x is assigned a value, it becomes a proposition. The statement “This sentence is false” is not a proposition as it contains a self-reference that contradicts itself. Clearly, if the statement is true it is false, and if it is false it is true.

A propositional variable may be used to stand for a proposition (e.g. let the variable P stand for the proposition “ $2 + 2 = 4$ ” which is a true proposition). A propositional variable takes the value true or false. The negation of a proposition P (denoted $\neg P$) is the proposition that is true if and only if P is false, and is false if and only if P is true.

A well-formed formula (*WFF*) in propositional logic is a syntactically correct formula created according to the syntactic rules of the underlying calculus. A well-formed formula is built up from variables, constants, terms and logical connectives such as conjunction (and), disjunction (or), implication (if ... then ...), equivalence (if and only if) and negation. A distinguished subset of these well-formed formulae are the *axioms* of the calculus, and there are *rules of inference* that allow the truth of new formulae to be derived from the axioms and from formulae that have already demonstrated to be true in the calculus.

A formula in propositional calculus may contain several propositional variables, and the truth or falsity of the individual variables needs to be known prior to determining the truth or falsity of the logical formula.

Each propositional variable has two possible values, and a formula with n -propositional variables has 2^n values associated with the n -propositional variables. The set of values associated with the n variables may be used to derive a truth table with 2^n rows and $n + 1$ columns. Each row gives each of the 2^n truth-values that the n variables may take, and column $n + 1$ gives the result of the logical expression for that set of values of the propositional variables. For example, the propositional formula W defined in the truth table above (Table 6.1) has two propositional variables A and B , with $2^2 = 4$ rows for each of the values that the two propositional variables may take. There are $2 + 1 = 3$ columns with W defined in the third column.

A rich set of connectives is employed in the calculus to combine propositions and to build up the well-formed formulae. This includes the conjunction of two propositions ($A \wedge B$); the disjunction of two propositions ($A \vee B$); and the implication of two propositions ($A \rightarrow B$). These connectives allow compound propositions to be formed, and the truth of the compound propositions is determined from the truth-values of its constituent propositions and the rules associated with the

Table 6.1 Truth table for formula W

A	B	$W(A, B)$
T	T	T
T	F	F
F	T	F
F	F	T

logical connectives. The meaning of the logical connectives is given by truth tables.¹

Mathematical logic is concerned with inference, and it involves proceeding in a methodical way from the axioms and using the rules of inference to derive further truths.

The rules of inference allow new propositions to be deduced from an existing set of propositions. A valid argument (or deduction) is truth preserving; that is, for a valid logical argument, if the set of premises is true, then the conclusion (i.e. the deduced proposition) will also be true. The rules of inference include rules such as *modus ponens*, and this rule states that given the truths of the proposition A , and the proposition $A \rightarrow B$, then the truth of proposition B may be deduced.

The propositional calculus is employed in reasoning about propositions, and it may be applied to formalize arguments in natural language. *Boolean algebra* is used in computer science, and it is named after George Boole, who was the first professor of mathematics at Queens College, Cork.² His symbolic logic (discussed in Chap. 5) is the foundation for modern computing.

6.2.1 Truth Tables

Truth tables give operational definitions of the most important logical connectives, and they provide a mechanism to determine the truth-values of more complicated compound expressions. Compound expressions are formed from propositions and connectives, and the truth-values of a compound expression containing several propositional variables are determined from the underlying propositional variables and the logical connectives.

The conjunction of A and B (denoted $A \wedge B$) is true if and only if both A and B are true, and is false in all other cases (Table 6.2). The disjunction of two propositions A and B (denoted $A \vee B$) is true if at least one of A and B are true, and false in all other cases (Table 6.3). The disjunction operator is known as the “*inclusive or*” operator as it is also true when both A and B are true; there is also an *exclusive or* operator that is true exactly when one of A or B is true, and is false otherwise.

Example 6.1 Consider proposition A given by “An orange is a fruit” and the proposition B given by “ $2 + 2 = 5$ ”, then A is true and B is false. Therefore,

¹Basic truth tables were first used by Frege and developed further by Post and Wittgenstein.

²This institution is now known as University College Cork and has approximately 20,000 students.

Table 6.2 Conjunction

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

Table 6.3 Disjunction

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

- (i) $A \wedge B$ (i.e. An orange is a fruit and $2 + 2 = 5$) is false.
(ii) $A \vee B$ (i.e. An orange is a fruit or $2 + 2 = 5$) is true.

The implication operation ($A \rightarrow B$) is true if whenever A is true means that B is also true and also whenever A is false (Table 6.4). It is equivalent (as shown by a truth table) to $\neg A \vee B$. The equivalence operation ($A \leftrightarrow B$) is true whenever both A and B are true, or whenever both A and B are false (Table 6.5).

The not operator (\neg) is a unary operator (i.e. it has one argument) and is such that $\neg A$ is true when A is false, and is false when A is true (Table 6.6).

Example 6.2 Consider proposition A given by “Jaffa cakes are biscuits” and the proposition B given by “ $2 + 2 = 5$ ”, then A is true and B is false. Therefore,

- (i) $A \rightarrow B$ (i.e. Jaffa cakes are biscuits implies $2 + 2 = 5$) is false.
(ii) $A \leftrightarrow B$ (i.e. Jaffa cakes are biscuits is equivalent to $2 + 2 = 5$) is false.
(iii) $\neg B$ (i.e. $2 + 2 \neq 5$) is true.

Creating a Truth Table

The truth table for a well-formed formula $W(P_1, P_2, \dots, P_n)$ is a table with 2^n rows and $n + 1$ columns. Each row lists a different combination of truth-values of the propositions P_1, P_2, \dots, P_n followed by the corresponding truth-value of W .

The example above (Table 6.7) gives the truth table for a formula W with three propositional variables (meaning that there are $2^3 = 8$ rows in the truth table).

Table 6.4 Implication

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

Table 6.5 Equivalence

A	B	$A \leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

Table 6.6 NOT operation

A	$\neg A$
T	F
F	T

Table 6.7 Truth table for W
(P, Q, R)

P	Q	R	$W(P, Q, R)$
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

6.2.2 Properties of Propositional Calculus

There are many well-known properties of the propositional calculus such as the commutative, associative and distributive properties. These ease the evaluation of complex expressions and allow logical expressions to be simplified.

The *commutative property* holds for the conjunction and disjunction operators, and it states that the order of evaluation of the two propositions may be reversed without affecting the resulting truth-value, i.e.

$$A \wedge B = B \wedge A$$

$$A \vee B = B \vee A$$

The *associative property* holds for the conjunction and disjunction operators. This means that order of evaluation of a subexpression does not affect the resulting truth-value, i.e.

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$

$$(A \vee B) \vee C = A \vee (B \vee C)$$

The conjunction operator *distributes* over the disjunction operator and vice versa.

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

The result of the logical conjunction of two propositions is false if one of the propositions is false (irrespective of the value of the other proposition).

$$A \wedge F = F \wedge A = F$$

The result of the logical disjunction of two propositions is true if one of the propositions is true (irrespective of the value of the other proposition).

$$A \vee T = T \vee A = T$$

The result of the logical disjunction of two propositions, where one of the propositions is known to be false, is given by the truth-value of the other proposition. That is, the Boolean value “F” acts as the identity for the disjunction operation.

$$A \vee F = A = F \vee A$$

The result of the logical conjunction of two propositions, where one of the propositions is known to be true, is given by the truth-value of the other proposition. That is, the Boolean value “T” acts as the identity for the conjunction operation.

$$A \wedge T = A = T \wedge A$$

The \wedge and \vee operators are *idempotent*. That is, when the arguments of the conjunction or disjunction operator are the same proposition A , the result is A . The idempotent property allows expressions to be simplified.

$$A \wedge A = A$$

$$A \vee A = A$$

The *law of the excluded middle* is a fundamental property of the propositional calculus. It states that a proposition A is either true or false; that is, there is no third logical value.

$$A \vee \neg A$$

Table 6.8 Tautology $B \vee \neg B$

B	$\neg B$	$B \vee \neg B$
T	F	T
F	T	T

We mentioned earlier that $A \rightarrow B$ is logically equivalent to $\neg A \vee B$ (same truth table), and clearly $\neg A \vee B$ is equivalent to $\neg A \vee \neg \neg B$, which is equivalent to $\neg \neg B \vee \neg A$ which is logically equivalent to $\neg B \rightarrow \neg A$. In other words, $A \rightarrow B$ is logically equivalent to $\neg B \rightarrow \neg A$, and this is known as the *contrapositive*.

De Morgan was a contemporary of Boole in the nineteenth century, and the following law is known as De Morgan's law.

$$\neg(A \vee B) \equiv \neg A \vee \neg B$$

$$\neg(A \wedge B) \equiv \neg A \wedge \neg B$$

Certain well-formed formulae are true for all values of their constituent variables. This can be seen from the truth table when the last column of the truth table consists entirely of true values.

A proposition that is true for all values of its constituent propositional variables is known as a *tautology*. An example of a tautology is the proposition $A \vee \neg A$ (Table 6.8)

A proposition that is false for all values of its constituent propositional variables is known as a *contradiction*. An example of a contradiction is the proposition $A \wedge \neg A$.

6.2.3 Proof in Propositional Calculus

Logic enables further truths to be derived from existing truths by rules of inference that are truth preserving. Propositional calculus is both *complete* and *consistent*. The completeness property means that all true propositions are deducible in the calculus, and the consistency property means that there is no formula A such that both A and $\neg A$ are deducible in the calculus.

An argument in propositional logic consists of a sequence of formulae that are the premises of the argument and a further formula that is the conclusion of the argument. One elementary way to see if the argument is valid is to produce a truth table to determine if the conclusion is true whenever all of the premises are true.

Consider a set of premises P_1, P_2, \dots, P_n and conclusion Q . Then, to determine if the argument is valid using a truth table involves adding a column in the truth table for each premise P_1, P_2, \dots, P_n , and then to identify the rows in the truth table for which these premises are all true. The truth-value of the conclusion Q is examined in each of these rows, and if Q is true for each case for which P_1, P_2, \dots, P_n are all true, then the argument is valid. This is equivalent to $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$ is a tautology.

An alternate approach to proof with truth tables is to assume the negation of the desired conclusion (i.e. $\neg Q$) and to show that the premises and the negation of the conclusion result in a contradiction (i.e. $P_1 \wedge P_2 \wedge \dots \wedge P_n \wedge \neg Q$) is a contradiction.

The use of truth tables becomes cumbersome when there are a large number of variables involved, as there are 2^n truth table entries for n -propositional variables.

Procedure for Proof by Truth Table

- (i) Consider argument P_1, P_2, \dots, P_n with conclusion Q .
- (ii) Draw truth table with column in truth table for each premise P_1, P_2, \dots, P_n .
- (iii) Identify rows in truth table for when these premises are all true.
- (iv) Examine truth-value of Q for these rows.
- (v) If Q is true for each case that P_1, P_2, \dots, P_n are true, then the argument is valid.
- (vi) That is $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$ is a tautology.

Example 6.3 (Truth Tables) Consider the argument adapted from [Kel:97] and determine if it is valid.

If the pianist plays the concerto, then crowds will come if the prices are not too high.

If the pianist plays the concerto, then the prices will not be too high.

Therefore, if the pianist plays the concerto, then crowds will come.

Solution

We will adopt a common proof technique that involves showing that the negation of the conclusion is incompatible (inconsistent) with the premises, and from this we deduce that the conclusion must be true. First, we encode the argument in propositional logic:

Let P stand for “The pianist plays the concerto”; C stands for “Crowds will come”; and H stands for “Prices are too high”. Then, the argument may be expressed in propositional logic as:

$$\begin{array}{ll}
 P \rightarrow (\neg H \rightarrow C) & \text{(Premise 1)} \\
 P \rightarrow \neg H & \text{(Premise 2)} \\
 \dots & \\
 P \rightarrow C & \text{(Conclusion)}
 \end{array}$$

Then, we negate the conclusion $P \rightarrow C$ and check the consistency of $P \rightarrow (\neg H \rightarrow C) \wedge (P \rightarrow \neg H) \wedge \neg (P \rightarrow C)$ * using a truth table (Table 6.9).

It can be seen from the last column in the truth table that the negation of the conclusion is incompatible with the premises, and therefore it cannot be the case that the premises are true and the conclusion is false. Therefore, the conclusion

Table 6.9 Proof of argument with a truth table

P	C	H	$\neg H$	$\neg H \rightarrow C$	$P \rightarrow (\neg H \rightarrow C)$	$P \rightarrow \neg H$	$P \rightarrow C$	$\neg(P \rightarrow C)$	*
T	T	T	F	T	T	F	T	F	F
T	T	F	T	T	T	T	T	F	F
T	F	T	F	T	T	F	F	T	F
T	F	F	T	F	F	T	F	T	F
F	T	T	F	T	T	T	T	F	F
F	T	F	T	T	T	T	T	F	F
F	F	T	F	T	T	T	T	F	F
F	F	F	T	F	T	T	T	F	F

must be true whenever the premises are true, and we conclude that the argument is valid.

Logical Equivalence and Logical Implication

The laws of mathematical reasoning are truth preserving and are concerned with deriving further truths from existing truths. Logical reasoning is concerned with moving from one line in mathematical argument to another and involves deducing the truth of another statement Q from the truth of P .

The statement Q may be in some sense be logically equivalent to P , and this allows the truth of Q to be immediately deduced. In other cases, the truth of P is sufficiently strong to deduce the truth of Q ; in other words, P logically implies Q . This leads naturally to a discussion of the concepts of logical equivalence ($W_1 \equiv W_2$) and logical implication ($W_1 \vdash W_2$).

Logical Equivalence

Two well-formed formulae W_1 and W_2 with the same propositional variables ($P, Q, R \dots$) are logically equivalent ($W_1 \equiv W_2$) if they are always simultaneously true or false for any given truth-values of the propositional variables.

If two well-formed formulae are logically equivalent, then it does not matter which of W_1 and W_2 is used, and $W_1 \leftrightarrow W_2$ is a tautology. In Table 6.10, we see that $P \wedge Q$ is logically equivalent to $\neg(\neg P \vee \neg Q)$.

Logical Implication

For two well-formed formulae W_1 and W_2 with the same propositional variables ($P, Q, R \dots$), W_1 logically implies W_2 ($W_1 \vdash W_2$) if any assignment to the propositional variables which makes W_1 true also makes W_2 true (Table 6.11). That is, $W_1 \rightarrow W_2$ is a tautology.

Example 6.4 Show by truth tables that $(P \wedge Q) \vee (Q \wedge \neg R) \vdash (Q \vee R)$.

The formula $(P \wedge Q) \vee (Q \wedge \neg R)$ is true on rows 1, 2 and 6, and formula $(Q \vee R)$ is also true on these rows. Therefore, $(P \wedge Q) \vee (Q \wedge \neg R) \vdash (Q \vee R)$.

Table 6.10 Logical equivalence of two WFFs

P	Q	$P \wedge Q$	$\neg P$	$\neg Q$	$\neg P \vee \neg Q$	$\neg(\neg P \vee \neg Q)$
T	T	T	F	F	F	T
T	F	F	F	T	T	F
F	T	F	T	F	T	F
F	F	F	T	T	T	F

Table 6.11 Logical implication of two WFFs

PQP	$(P \wedge Q) \vee (Q \wedge \neg R)$	$Q \vee R$
TTT	T	T
TTF	T	T
TFT	F	T
TFF	F	F
FTT	F	T
FTF	T	T
FFT	F	T
FFF	F	F

Show by truth tables that $(P \wedge Q) \vee (Q \wedge \neg R) \vdash (Q \vee R)$

6.2.4 Semantic Tableaux in Propositional Logic

We showed in Example 6.3 how truth tables may be used to demonstrate the validity of a logical argument. However, the problem with truth tables is that they can get extremely large very quickly (as the size of the table is 2^n where n is the number of propositional variables), and so in this section we will consider an alternate approach known as semantic tableaux.

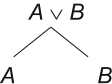
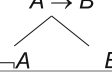
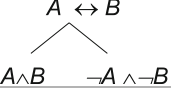
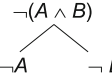
The basic idea of semantic tableaux is to determine if it is possible for a conclusion to be false when all of the premises are true. If this is not possible, then the conclusion must be true when the premises are true, and so the conclusion is *semantically entailed* by the premises. The method of semantic tableaux was developed by the Dutch logician, Evert Willem Beth, and the technique exposes inconsistencies in a set of logical formulae by identifying conflicting logical expressions.

We present a short summary of the rules of semantic tableaux in Table 6.12, and we then proceed to provide a proof for Example 6.3 using semantic tableaux instead of a truth table.

Whenever a logical expression A and its negation $\neg A$ appear in a branch of the tableau, then an inconsistency has been identified in that branch, and the branch is said to be *closed*. If all of the branches of the semantic tableaux are closed, then the logical propositions from which the tableau was formed are mutually inconsistent and cannot be true together.

The method of proof is to negate the conclusion, and to show that all branches in the semantic tableau are closed and that therefore it is not possible for the premises of the argument to be true and for the conclusion to be false. Therefore, the argument is valid and the conclusion follows from the premises.

Table 6.12 Rules of semantic tableaux

Rule No.	Definition	Description
1	$A \wedge B$ A B	If $A \wedge B$ is true, then both A and B are true and may be added to the branch containing $A \wedge B$
2.	$A \vee B$ 	If $A \vee B$ is true, then either A or B is true, and we add two new branches to the tableaux, one containing A and one containing B
3.	$A \rightarrow B$ 	If $A \rightarrow B$ is true, then either $\neg A$ or B is true, and we add two new branches to the tableaux, one containing $\neg A$ and one containing B
4.	$A \leftrightarrow B$ 	If $A \leftrightarrow B$ is true, then either $A \wedge B$ or $\neg A \wedge \neg B$ is true, and we add two new branches, one containing $A \wedge B$ and one containing $\neg A \wedge \neg B$
5.	$\neg\neg A$ A	If $\neg\neg A$ is true, then A may be added to the branch containing $\neg\neg A$
6.	$\neg(A \wedge B)$ 	If $\neg(A \wedge B)$ is true, then either $\neg A \vee \neg B$ is true, and we add two new branches to the tableaux, one containing $\neg A$ and one containing $\neg B$
7.	$\neg(A \vee B)$ $\neg A$ $\neg B$	If $\neg(A \vee B)$ is true, then both $\neg A \wedge \neg B$ are true and may be added to the branch containing $\neg(A \vee B)$
8.	$\neg(A \rightarrow B)$ A $\neg B$	If $\neg(A \rightarrow B)$ is true, then both $A \wedge \neg B$ are true and may be added to the branch containing $\neg(A \rightarrow B)$

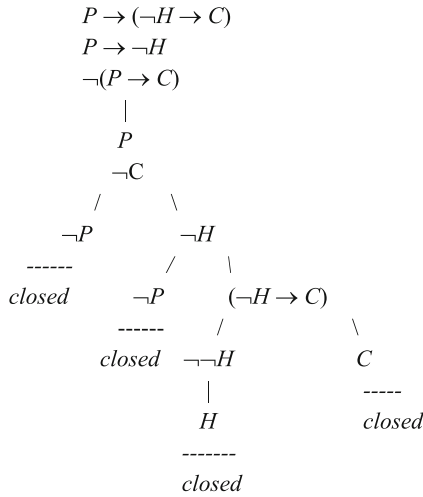
Example 6.5 (Semantic Tableaux) Perform the proof for Example 6.3 using semantic tableaux.

Solution

We formalized the argument previously as

- (Premise 1) $P \rightarrow (\neg H \rightarrow C)$
- (Premise 2) $P \rightarrow \neg H$
- (Conclusion) $P \rightarrow C$

We negate the conclusion to get $\neg(P \rightarrow C)$, and we show that all branches in the semantic tableau are closed and that therefore it is not possible for the premises of the argument to be true and for the conclusion to be false. Therefore, the argument is valid, and the truth of the conclusion follows from the truth of the premises.



We have showed that all branches in the semantic tableau are closed and that therefore it is not possible for the premises of the argument to be true and for the conclusion to be false. Therefore, the argument is valid as required.

6.2.5 Natural Deduction

The German mathematician, Gerhard Gentzen (Fig. 6.1), developed a method for logical deduction known as “*Natural Deduction*”, and his formal approach to natural deduction aimed to be as close as possible to natural reasoning. Gentzen worked as an assistant to David Hilbert at the University of Göttingen, and he died of malnutrition in Prague towards the end of the Second World War.

Natural deduction includes rules for \wedge , \vee , \rightarrow introduction and elimination and also for *reductio ad absurdum*. There are ten inference rules in the natural deduction system, and they include two inference rules for each of the five logical operators \wedge , \vee , \neg , \rightarrow and \leftrightarrow (an introduction rule and an elimination rule), and the rules are defined in Table 6.13:

Natural deduction may be employed in logical reasoning and is described in detail in [Gri:81, Kel:97].

Fig. 6.1 Gerhard Gentzen



Table 6.13 Natural deduction rules

Rule	Definition	Description
\wedge I	$\frac{P_1, P_2, \dots, P_n}{P_1 \wedge P_2 \wedge \dots \wedge P_n}$	Given the truth of propositions P_1, P_2, \dots, P_n , then the truth of the conjunction $P_1 \wedge P_2 \wedge \dots \wedge P_n$ follows. This rule shows how conjunction can be introduced
\wedge E	$\frac{P_1 \wedge P_2 \wedge \dots \wedge P_n}{P_i}$ where $i \in \{1, \dots, n\}$	Given the truth the conjunction $P_1 \wedge P_2 \wedge \dots \wedge P_n$, then the truth of proposition P_i follows. This rule shows how a conjunction can be eliminated
\vee I	$\frac{P_i}{P_1 \vee P_2 \vee \dots \vee P_n}$	Given the truth of propositions P_i , then the truth of the disjunction $P_1 \vee P_2 \vee \dots \vee P_n$ follows. This rule shows how a disjunction can be introduced
\vee E	$\frac{P_1 \vee \dots \vee P_n, P_1 \rightarrow E, \dots, P_n \rightarrow E}{E}$	Given the truth of the disjunction $P_1 \vee P_2 \vee \dots \vee P_n$ and that each disjunct implies E, then the truth of E follows. This rule shows how a disjunction can be eliminated
\rightarrow I	$\frac{\text{From } P_1, P_2, \dots, P_n \text{ infer } P}{(P_1 \wedge P_2 \wedge \dots \wedge P_n) \rightarrow P}$	This rule states that if we have a theorem that allows P to be inferred from the truth of premises P_1, P_2, \dots, P_n (or previously proved), then we can deduce $(P_1 \wedge P_2 \wedge \dots \wedge P_n) \rightarrow P$. This is known as the <i>deduction theorem</i>
\rightarrow E	$\frac{P_i \rightarrow P_j, P_i}{P_j}$	This rule is known as <i>modus ponens</i> . The consequence of an implication follows if the antecedent is true (or has been previously proved)
\equiv I	$\frac{P_i \rightarrow P_j, P_j \rightarrow P_i}{P_i \leftrightarrow P_j}$	If proposition P_i implies proposition P_j and vice versa, then they are equivalent (i.e. $P_i \leftrightarrow P_j$)
\equiv E	$\frac{P_i \leftrightarrow P_j}{P_i \rightarrow P_j, P_j \rightarrow P_i}$	If proposition P_i is equivalent to proposition P_j , then proposition P_i implies proposition P_j and vice versa
\neg I	$\frac{\text{From } P \text{ infer } P_1 \wedge \neg P_1}{\neg P}$	If the proposition P allows a contradiction to be derived, then $\neg P$ is deduced. This is an example of a <i>proof by contradiction</i>
\neg E	$\frac{\text{From } \neg P \text{ infer } P_1 \wedge \neg P_1}{P}$	If the proposition $\neg P$ allows a contradiction to be derived, then P is deduced. This is an example of a <i>proof by contradiction</i>

6.2.6 Sketch of Formalization of Propositional Calculus

Truth tables provide an informal approach to proof, and the proof is provided in terms of the truth-values of the propositions and the meaning of the logical connectives. The formalization of propositional logic includes the definition of an alphabet of symbols and well-formed formulae of the calculus, the axioms of the calculus and rules of inference for logical deduction.

The deduction of a new formulae Q is via a sequence of well-formed formulae P_1, P_2, \dots, P_n (where $P_n = Q$) such that each P_i is either an axiom, a hypothesis, or deducible from an earlier pair of formula P_j, P_k , (where P_k is of the form $P_j \rightarrow P_i$) and modus ponens. *Modus ponens* is a rule of inference that states that given propositions A , and $A \rightarrow B$, then proposition B may be deduced. The deduction of a formula Q from a set of hypothesis H is denoted by $H \vdash Q$, and where Q is deducible from the axioms alone this is denoted by $\vdash Q$.

The *deduction theorem* of propositional logic states that if $H \cup \{P\} \vdash Q$, then $H \vdash P \rightarrow Q$, and the converse of the theorem is also true; that is, if $H \vdash P \rightarrow Q$, then $H \cup \{P\} \vdash Q$. Formalism (this approach was developed by the German mathematician, David Hilbert) allows reasoning about symbols according to rules, and to derive theorems from formulae irrespective of the meanings of the symbols and formulae.

Propositional calculus is *sound* (i.e. any theorem derived using the Hilbert approach is true). Further, the calculus is also *complete*, and every tautology has a proof (i.e. it is a theorem in the formal system). The propositional calculus is *consistent* (i.e. it is not possible that both the well-formed formula A and $\neg A$ are deducible in the calculus).

Propositional calculus is *decidable*; that is, there is an algorithm (e.g. a truth table) to determine for any well-formed formula A whether A is a theorem of the formal system. The Hilbert style system is slightly cumbersome in conducting proof and is quite different from the normal use of logic in mathematical deduction.

6.2.7 Applications of Propositional Calculus

Propositional calculus may be employed in reasoning with arguments in natural language. First, the premises and conclusion of the argument are identified and formalized into propositions. Propositional logic is then employed to determine if the conclusion is a valid deduction from the premises.

Consider, for example, the following argument that aims to prove that Superman does not exist.

If Superman were able and willing to prevent evil, he would do so. If Superman were unable to prevent evil he would be impotent; if he were unwilling to prevent evil he would be malevolent; Superman does not prevent evil. If superman exists he is neither malevolent nor impotent; therefore Superman does not exist.

First, letters are employed to represent the propositions as follows:

- a* Superman is able to prevent evil
- w* Superman is willing to prevent evil
- i* Superman is impotent
- m* Superman is malevolent
- p* Superman prevents evil
- e* Superman exists

Then, the argument above is formalized in propositional logic as follows:

Premises

- $P_1 \quad (a \wedge w) \rightarrow p$
- $P_2 \quad (\neg a \rightarrow i) \wedge (\neg w \rightarrow m)$
- $P_3 \quad \neg p$
- $P_4 \quad e \rightarrow \neg i \wedge \neg m$

Conclusion $\neg e \quad (P_1 \wedge P_2 \wedge P_3 \wedge P_4 \vdash \neg e)$

Proof that Superman does not exist

1.	$a \wedge w \rightarrow p$	<i>Premise 1</i>
2.	$(\neg a \rightarrow i) \wedge (\neg w \rightarrow m)$	<i>Premise 2</i>
3.	$\neg p$	<i>Premise 3</i>
4.	$e \rightarrow (\neg i \wedge \neg m)$	<i>Premise 4</i>
5.	$\neg p \rightarrow \neg(a \wedge w)$	<i>1, Contrapositive</i>
6.	$\neg(a \wedge w)$	<i>3,5 Modus Ponens</i>
7.	$\neg a \vee \neg w$	<i>6, De Morgan's law</i>
8.	$\neg(\neg i \wedge \neg m) \rightarrow \neg e$	<i>4, Contrapositive</i>
9.	$i \vee m \rightarrow \neg e$	<i>8, De Morgan's law</i>
10.	$(\neg a \rightarrow i)$	<i>2, \wedge Elimination</i>
11.	$(\neg w \rightarrow m)$	<i>2, \wedge Elimination</i>
12.	$\neg \neg a \vee i$	<i>10, $A \rightarrow B$ equivalent to $\neg A \vee B$</i>
13.	$\neg \neg a \vee i \vee m$	<i>11, \vee Introduction</i>
14.	$\neg \neg a \vee (i \vee m)$	
15.	$\neg a \rightarrow (i \vee m)$	<i>14, $A \rightarrow B$ equivalent to $\neg A \vee B$</i>
16.	$\neg \neg w \vee m$	<i>11, $A \rightarrow B$ equivalent to $\neg A \vee B$</i>
17.	$\neg \neg w \vee (i \vee m)$	
18.	$\neg w \rightarrow (i \vee m)$	<i>17, $A \rightarrow B$ equivalent to $\neg A \vee B$</i>
19.	$(i \vee m)$	<i>7, 15, 18 \vee Elimination</i>
20.	$\neg e$	<i>9, 19 Modus Ponens</i>

Second Proof

1.	$\neg p$	P_3
2.	$\neg(a \wedge w) \vee p$	$P_1 (A \rightarrow B \equiv \neg A \vee B)$
3.	$\neg(a \wedge w)$	1,2 $A \vee B, \neg B \vdash A$
4.	$\neg a \vee \neg w$	3, De Morgan's law
5.	$(\neg a \rightarrow i)$	$P_2 (\wedge\text{-Elimination})$
6.	$\neg a \rightarrow i \vee m$	5, $x \rightarrow y \vdash x \rightarrow y \vee z$
7.	$(\neg w \rightarrow m)$	$P_2 (\wedge\text{-Elimination})$
8.	$\neg w \rightarrow i \vee m$	7, $x \rightarrow y \vdash x \rightarrow y \vee z$
9.	$(\neg a \vee \neg w) \rightarrow (i \vee m)$	8, $x \rightarrow z, y \rightarrow z \vdash x \vee y \rightarrow z$
10.	$(i \vee m)$	4,9 Modus Ponens
11.	$e \rightarrow \neg(i \vee m)$	P_4 (De Morgan's law)
12.	$\neg e \vee \neg(i \vee m)$	11, $(A \rightarrow B \equiv \neg A \vee B)$
13.	$\neg e$	10, 12 $A \vee B, \neg B \vdash A$

Therefore, the conclusion that Superman does not exist is a valid deduction from the given premises.

6.2.8 Limitations of Propositional Calculus

The propositional calculus deals with propositions only. It is incapable of dealing with the syllogism “All Greeks are mortal; Socrates is a Greek; therefore Socrates is mortal”. This would be expressed in propositional calculus as three propositions A , B and therefore C , where A stands for “All Greeks are mortal”, B stands for “Socrates is a Greek” and C stands for “Socrates is mortal”. Propositional logic does not allow the conclusion that all Greeks are mortal to be derived from the two premises.

Predicate calculus deals with these limitations by employing variables and terms, and using universal and existential quantification to express that a particular property is true of all (or at least one) values of a variable. Predicate calculus is discussed in the next section.

6.3 Predicate Calculus

Predicate logic is a richer system than propositional logic, and it allows complex facts about the world to be represented. It allows new facts about the world to be derived in a way that guarantees that if the initial premises are true, then the conclusions are true. Predicate calculus consists of predicates, variables, constants and quantifiers.

A *predicate* is a characteristic or property that an object can have, and we are predicating some property of the object. For example, “*Socrates is a Greek*” could be expressed as $G(s)$, with capital letters standing for predicates and small letters standing for objects. A predicate may include variables, and a statement with a variable becomes a proposition once the variables are assigned values. For example, $G(x)$ states that the variable x is a Greek, whereas $G(s)$ is an assignment of values to x . The set of values that the variables may take is termed the universe of discourse, and the variables take values from this set.

Predicate calculus employs quantifiers to express properties such as all members of the domain have a particular property: e.g. $(\forall x)P(x)$, or that there is at least one member that has a particular property: e.g. $(\exists x)P(x)$. These are referred to as the *universal and existential quantifiers*.

The syllogism “All Greeks are mortal; Socrates is a Greek; therefore Socrates is mortal” may be easily expressed in predicate calculus by:

$$\begin{array}{l} (\forall x)(G(x) \rightarrow M(x)) \\ G(s) \\ \hline M(s) \end{array}$$

In this example, the predicate $G(x)$ stands for x is a Greek and the predicate $M(x)$ stands for x is mortal. The formula $G(x) \rightarrow M(x)$ states that if x is a Greek, then x is mortal, and the formula $(\forall x)(G(x) \rightarrow M(x))$ states for any x that if x is a Greek, then x is mortal. The formula $G(s)$ states that Socrates is a Greek, and the formula $M(s)$ states that Socrates is mortal.

Example 6.6 (Predicates) A predicate may have one or more variables. A predicate that has only one variable (i.e. a unary or one-place predicate) is often related to sets; a predicate with two variables (a two-place predicate) is a relation; and a predicate with n variables (a n -place predicate) is a n -ary relation. Propositions do not contain variables, and so they are zero-place predicates. The following are examples of predicates:

- (i) The predicate $Prime(x)$ states that x is a prime number (with the natural numbers being the universe of discourse).
- (ii) $Lawyer(a)$ may stand for a is a lawyer.
- (iii) $Mean(m, x, y)$ states that m is the mean of x and y : i.e., $m = \frac{1}{2}(x+y)$.
- (iv) $LT(x, 6)$ states that x is less than 6.
- (v) $GT(x, \pi)$ states that x is greater than π (where π is the constant 3.14159).
- (vi) $GT(x, y)$ states that x is greater than y .
- (vii) $EQ(x, y)$ states that x is equal to y .
- (viii) $LE(x, y)$ states that x is less than or equal to y .
- (ix) $Real(x)$ states that x is a real number.
- (x) $Father(x, y)$ states that x is the father of y .
- (xi) $\neg(\exists x)(Prime(x) \wedge BE(x, 32, 36))$ states that there is no prime number between 32 and 36.

Universal and Existential Quantification

The universal quantifier is used to express a statement such as that all members of the domain have property P . This is written as $(\forall x)P(x)$ and expresses the statement that the property $P(x)$ is true for all x . Similarly, $(\forall x_1, x_2, \dots, x_n) P(x_1, x_2, \dots, x_n)$ states that property $P(x_1, x_2, \dots, x_n)$ is true for all x_1, x_2, \dots, x_n . Clearly, the predicate $(\forall x) P(a, b)$ is identical to $P(a, b)$ since it contains no variables, and the predicate $(\forall y \in \mathbb{N}) (x \leq y)$ is true if $x = 1$ and false otherwise.

The existential quantifier states that there is at least one member in the domain of discourse that has property P . This is written as $(\exists x)P(x)$, and the predicate $(\exists x_1, x_2, \dots, x_n) P(x_1, x_2, \dots, x_n)$ states that there is at least one value of (x_1, x_2, \dots, x_n) such that $P(x_1, x_2, \dots, x_n)$ is true.

Example 6.7 (Quantifiers)

- (i) $(\exists p) (Prime(p) \wedge p > 1,000,000)$ is true

It expresses the fact that there is at least one prime number greater than a million, which is true as there are an infinite number of primes.

- (ii) $(\forall x) (\exists y) x < y$ is true

This predicate expresses the fact that given any number x , we can always find a larger number: e.g. take $y = x + 1$.

(iii) $(\exists y)(\forall x) x < y$ is false

This predicate expresses the statement that there is a natural number y such that all natural numbers are less than y . Clearly, this statement is false since there is no largest natural number, and so the predicate $(\exists y)(\forall x) x < y$ is false.

Comment 6.1 It is important to be careful with the order in which quantifiers are written, as the meaning of a statement may be completely changed by the simple transposition of two quantifiers.

The well-formed formulae in the predicate calculus are built from terms and predicates, and the rules for building the formulae are sketched in Sect. 6.3.1. Examples of well-formed formulae include:

$$\begin{aligned} &(\forall x)(x > 2) \\ &(\exists x)x^2 = 2 \\ &(\forall x)(x > 2 \wedge x < 10) \\ &(\forall x)(\exists y)x^2 = y \\ &(\forall x)(\exists y)Love(y, x) \quad (\text{everyone is loved by some one}) \\ &(\exists y)(\forall x)Love(y, x) \quad (\text{some one loves every one}) \end{aligned}$$

The formula $(\forall x)(x > 2)$ states that every x is greater than the constant 2; $(\exists x)x^2 = 2$ states that there is an x that is the square root of 2; $(\forall x)(\exists y)x^2 = y$ states that for every x , there is a y such that the square of x is y .

6.3.1 Sketch of Formalization of Predicate Calculus

The formalization of predicate calculus includes the definition of an alphabet of symbols (including constants and variables), the definition of function and predicate letters, logical connectives and quantifiers. This leads to the definitions of the terms and well-formed formulae of the calculus.

The predicate calculus is built from an alphabet of constants, variables, function letters, predicate letters and logical connectives (including the logical connectives discussed in propositional logic, and the universal and existential quantifiers).

The definition of terms and well-formed formulae specifies the syntax of the predicate calculus, and the set of well-formed formulae gives the language of the calculus. The terms and well-formed formulae are built from the symbols, and these symbols are not given meaning in the formal definition of the syntax.

The language defined by the calculus needs to be given an *interpretation* in order to give meaning to the terms and formulae of the calculus. The interpretation needs to define the domain of values of the constants and variables, and provide meaning to the function letters, the predicate letters and the logical connectives.

Terms are built from constants, variables and function letters. A constant or variable is a term, and if t_1, t_2, \dots, t_k are terms, then $f_i^k(t_1, t_2, \dots, t_k)$ is a term (where f_i^k is a k -ary function letter). Examples of terms include:

x^2 where x is a variable and square is a 1-ary function letter
 x^2+y^2 where $x^2 + y^2$ is shorthand for the function $\text{add}(\text{square}(x), \text{square}(y))$ where add is a 2-ary function letter and square is a 1-ary function letter.

The well-formed formulae are built from terms as follows. If P_i^k is a k -ary predicate letter, t_1, t_2, \dots, t_k are terms, then $P_i^k(t_1, t_2, \dots, t_k)$ is a well-formed formula. If A and B are well-formed formulae, then so are $\neg A, A \wedge B, A \vee B, A \rightarrow B, A \leftrightarrow B, (\forall x)A$ and $(\exists x)A$.

There is a set of axioms for predicate calculus and two rules of inference used for the deduction of new formulae from the existing axioms and previously deduced formulae. The deduction of a new formula Q is via a sequence of well-formed formulae P_1, P_2, \dots, P_n (where $P_n = Q$) such that each P_i is either an axiom, a hypothesis, or deducible from one or more of the earlier formulae in the sequence.

The two rules of inference are *modus ponens* and *generalization*. Modus ponens is a rule of inference that states that given predicate formulae A , and $A \rightarrow B$, then the predicate formula B may be deduced. Generalization is a rule of inference that states that given predicate formula A , then the formula $(\forall x)A$ may be deduced where x is any variable.

The deduction of a formula Q from a set of hypothesis H is denoted by $H \vdash Q$, and where Q is deducible from the axioms alone this is denoted by $\vdash Q$. The *deduction theorem* states that if $H \cup \{P\} \vdash Q$, then $H \vdash P \rightarrow Q$ ³ and the converse of the theorem is also true; that is, if $H \vdash P \rightarrow Q$, then $H \cup \{P\} \vdash Q$.

The approach allows reasoning about symbols according to rules, and to derive theorems from formulae irrespective of the meanings of the symbols and formulae. Predicate calculus is *sound*; that is, any theorem derived using the approach is true, and the calculus is also *complete*.

Scope of Quantifiers

The scope of the quantifier $(\forall x)$ in the well-formed formula $(\forall x)A$ is A . Similarly, the scope of the quantifier $(\exists x)$ in the well-formed formula $(\exists x)B$ is B . The variable x that occurs within the scope of the quantifier is said to be a *bound variable*. If a variable is not within the scope of a quantifier, it is *free*.

Example 6.8 (Scope of Quantifiers)

- (i) x is free in the well-formed formula $\forall y (x^2 + y > 5)$.
- (ii) x is bound in the well-formed formula $\forall x (x^2 > 2)$.

³This is stated more formally that if $H \cup \{P\} \vdash Q$ by a deduction containing no application of generalization to a variable that occurs free in P , then $H \vdash P \rightarrow Q$.

A well-formed formula is *closed* if it has no free variables. The substitution of a term t for x in A can only take place only when no free variable in t will become bound by a quantifier in A through the substitution. Otherwise, the interpretation of A would be altered by the substitution.

A term t is free for x in A if no free occurrence of x occurs within the scope of a quantifier ($\forall y$) or ($\exists y$) where y is free in t . This means that the term t may be substituted for x without altering the interpretation of the well-formed formula A .

For example, suppose A is $\forall y (x^2 + y^2 > 2)$ and the term t is y , then t is not free for x in A as the substitution of t for x in A will cause the free variable y in t to become bound by the quantifier $\forall y$ in A , thereby altering the meaning of the formula to $\forall y (y^2 + y^2 > 2)$.

6.3.2 Interpretation and Valuation Functions

An *interpretation* gives meaning to a formula, and it consists of a *domain of discourse* and a *valuation function*. If the formula is a sentence (i.e. it does not contain any free variables), then the given interpretation of the formula is either true or false. If a formula has free variables, then the truth or falsity of the formula depends on the values given to the free variables. A formula with free variables essentially describes a relation say, $R(x_1, x_2, \dots, x_n)$ such that $R(x_1, x_2, \dots, x_n)$ is true if (x_1, x_2, \dots, x_n) is in relation R . If the formula is true irrespective of the values given to the free variables, then the formula is true in the interpretation.

A *valuation (meaning) function* gives meaning to the logical symbols and connectives. Thus associated with each constant c is a constant c_Σ in some universe of values Σ ; with each function symbol f of arity k , we have a function symbol f_Σ in Σ and $f_\Sigma : \Sigma^k \rightarrow \Sigma$; and for each predicate symbol P of arity k , we have a relation $P_\Sigma \subseteq \Sigma^k$. The *valuation function, in effect, gives the semantics of the language of the predicate calculus L .*

The truth of a predicate P is then defined in terms of the meanings of the terms, the meanings of the functions, predicate symbols and the normal meanings of the connectives.

Mendelson [Men:87] provides a technical definition of truth in terms of *satisfaction* (with respect to an interpretation M). Intuitively, a formula F is *satisfiable* if it is *true* (in the intuitive sense) for some assignment of the free variables in the formula F . If a formula F is satisfied for every possible assignment to the free variables in F , then it is *true* (in the technical sense) for the interpretation M . An analogous definition is provided for *false* in the interpretation M .

A formula is *valid* if it is true in every interpretation; however, as there may be an uncountable number of interpretations, it may not be possible to check this requirement in practice. M is said to be a model for a set of formulae if and only if every formula is true in M .

There is a distinction between proof theoretic and model theoretic approaches in predicate calculus. *Proof theoretic* is essentially syntactic, and there is a list of axioms with rules of inference. The theorems of the calculus are logically derived (i.e. $\vdash A$), and the logical truths are as a result of the syntax or form of the formulae, rather than the *meaning* of the formulae. *Model theoretic*, in contrast, is essentially semantic. The truth derives from the meaning of the symbols and connectives, rather than the logical structure of the formulae (written as $\vdash_M A$).

A calculus is *sound* if all of the logically valid theorems are true in the interpretation, i.e. proof theoretic \Rightarrow model theoretic. A calculus is *complete* if all the truths in an interpretation are provable in the calculus, i.e. model theoretic \Rightarrow proof theoretic. A calculus is *consistent* if there is no formula A such that $\vdash A$ and $\vdash \neg A$.

The predicate calculus is sound, complete and consistent. *Predicate calculus is not decidable*; that is, there is no algorithm to determine for any well-formed formula A whether A is a theorem of the formal system. The undecidability of the predicate calculus may be demonstrated by showing that if the predicate calculus is decidable, then the halting problem (of Turing machines) is solvable. The halting problem is discussed in Chap. 13 of [ORg:16b].

6.3.3 Properties of Predicate Calculus

The following are properties of the predicate calculus

- (i) $(\forall x)P(x) \equiv (\forall y)P(y)$
- (ii) $(\forall x)P(x) \equiv \neg(\exists x)\neg P(x)$
- (iii) $(\exists x)P(x) \equiv \neg(\forall x)\neg P(x)$
- (iv) $(\exists x)P(x) \equiv (\exists y)P(y)$
- (v) $(\forall x)(\forall y)P(x, y) \equiv (\forall y)(\forall x)P(x, y)$
- (vi) $(\exists x)(P(x) \vee Q(x)) \equiv (\exists x)P(x) \vee (\exists y)Q(y)$
- (vii) $(\forall x)(P(x) \wedge Q(x)) \equiv (\forall x)P(x) \wedge (\forall y)Q(y)$

6.3.4 Applications of Predicate Calculus

The predicate calculus may be employed to formally state the system requirements of a proposed system. It may be used to conduct formal proof to verify the presence or absence of certain properties in a specification.

It may also be employed to define piecewise defined functions such as $f(x, y)$ where $f(x, y)$ is defined by:

$$\begin{aligned} f(x, y) &= x^2 - y^2 && \text{where } x \leq 0 \wedge y < 0; \\ f(x, y) &= x^2 + y^2 && \text{where } x > 0 \wedge y < 0; \\ f(x, y) &= x + y && \text{where } x \geq 0 \wedge y = 0; \\ f(x, y) &= x - y && \text{where } x < 0 \wedge y = 0; \\ f(x, y) &= x + y && \text{where } x \leq 0 \wedge y > 0; \\ f(x, y) &= x^2 + y^2 && \text{where } x > 0 \wedge y > 0. \end{aligned}$$

The predicate calculus may be employed for program verification, and to show that a code fragment satisfies its specification. The statement that a program F is correct with respect to its precondition P and postcondition Q is written as $P\{F\}Q$. The objective of program verification is to show that if the precondition is true before execution of the code fragment, then this implies that the postcondition is true after execution of the code fragment.

A program fragment a is *partially correct* for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied and execution terminates, then the resulting state satisfies Q . Partial correctness is denoted by $P\{F\}Q$, and Hoare's axiomatic semantics is based on partial correctness. It requires proof that the postcondition is satisfied if the program terminates.

A program fragment a is *totally correct* for precondition P and postcondition Q , if and only if whenever a is executed in any state in which P is satisfied, then the execution terminates and the resulting state satisfies Q . It is denoted by $\{P\}F\{Q\}$, and Dijkstra's calculus of weakest preconditions is based on total correctness [Dij:76]. It is required to prove that if the precondition is satisfied, then the program terminates and the postcondition is satisfied.

6.3.5 Semantic Tableaux in Predicate Calculus

We discussed the use of semantic tableaux in determining the validity of arguments in propositional logic in Sect. 6.2.4, and its approach is to negate the conclusion of an argument and to show that this results in inconsistency with the premises of the argument.

The use of semantic tableaux is similar with predicate logic, except that there are some additional rules to consider. As before, the approach is to negate the conclusion and to show that this results in all branches of the semantic tableau being closed, and from this we deduce that the conclusion must be true.

The rules of semantic tableaux for propositional logic were presented in Table 6.12, and the additional rules specific to predicate logic are detailed in Table 6.14.

Example 6.9 (Semantic Tableaux) Show that the syllogism “All Greeks are mortal; Socrates is a Greek; therefore Socrates is mortal” is a valid argument in predicate calculus.

Solution

We expressed this argument previously as $(\forall x)(G(x) \rightarrow M(x)); G(s); M(s)$. Therefore, we negate the conclusion (i.e. $\neg M(s)$) and try to construct a closed tableau.

$$\begin{array}{l}
 (\forall x)(G(x) \rightarrow M(x)) \\
 G(s) \\
 \neg M(s). \\
 G(s) \rightarrow M(s) \qquad \text{Universal Instantiation} \\
 \quad \wedge \\
 \neg G(s) \quad M(s) \\
 \text{-----} \quad \text{-----} \\
 \text{closed} \qquad \text{closed}
 \end{array}$$

Therefore, as the tableau is closed, we deduce that the negation of the conclusion is inconsistent with the premises and that therefore the conclusion follows from the premises.

Example 6.10 (Semantic Tableaux) Determine whether the following argument is valid.

All lecturers are motivated.
 Anyone who is motivated and clever will teach well.
 Joanne is a clever lecturer.
 Therefore, Joanne will teach well.

Table 6.14 Extra rules of semantic tableaux (for predicate calculus)

Rule No.	Definition	Description
1	$(\forall x) A(x)$ $A(t)$ where t is a term	Universal instantiation
2.	$(\exists x) A(x)$ $A(t)$ where t is a term that has not been used in the derivation so far.	Rule of existential instantiation. The term “ t ” is often a constant “ a ”.
3.	$\neg(\forall x) A(x)$ $(\exists x) \neg A(x)$	
4.	$\neg(\exists x) A(x)$ $(\forall x)\neg A(x)$	

Solution

We encode the argument as follows

$L(x)$ stands for “ x is a lecturer”.

$M(x)$ stands for “ x is motivated”.

$C(x)$ stands for “ x is clever”.

$W(x)$ stands for “ x will teach well”.

We therefore wish to show that

$$(\forall x)(L(x) \rightarrow M(x)) \wedge (\forall x)((M(x) \wedge C(x)) \rightarrow W(x)) \wedge L(joanne) \wedge C(joanne) \models W(joanne)$$

Therefore, we negate the conclusion (i.e. $\neg W(joanne)$) and try to construct a closed tableau.

1. $(\forall x)(L(x) \rightarrow M(x))$
2. $(\forall x)((M(x) \wedge C(x)) \rightarrow W(x))$
3. $L(joanne)$
4. $C(joanne)$
5. $\neg W(joanne)$
6. $L(joanne) \rightarrow M(joanne)$ Universal Instantiation (line 1)

7. $(M(joanne) \wedge C(joanne)) \rightarrow W(joanne)$ Universal Instantiation (line 2)
 $\quad \quad \quad / \wedge$
8. $\frac{\neg L(joanne) \quad M(joanne)}{\text{Closed}}$ From line 6
9. $\frac{\neg(M(joanne) \wedge C(joanne)) \quad W(joanne)}{\text{Closed}}$ From line 7
 $\quad \quad \quad / \wedge$
10. $\frac{\neg M(joanne) \quad \neg C(joanne)}{\text{Closed} \quad \text{Closed}}$

Therefore, since the tableau is closed, we deduce that the argument is valid.

6.4 Review Questions

- Draw a truth table to show that $\neg(P \rightarrow Q) \equiv P \wedge \neg Q$.
- Translate the sentence "Execution of program P begun with $x < 0$ will not terminate" into propositional form.
- Prove the following theorems using the inference rules of natural deduction.
 - From b infer $b \vee \neg c$.
 - From $b \Rightarrow (c \wedge d)$, b infer d .
- Explain the difference between the universal and the existential quantifiers.
- Express the following statements in the predicate calculus:
 - All natural numbers are greater than 10.
 - There is at least one natural number between 5 and 10.
 - There is a prime number between 100 and 200.

6. Which of the following predicates are true?

- (a) $\forall i \in \{10, \dots, 50\}. i^2 < 2000 \wedge i < 100$
- (b) $\exists i \in \mathbb{N}. i > 5 \wedge i^2 = 25$
- (c) $\exists i \in \mathbb{N}. i^2 = 25$

7. Use semantic tableaux to show that $(A \rightarrow A) \vee (B \wedge \neg B)$ is true.

8. Determine if the following argument is valid.

If Pilar lives in Cork, she lives in Ireland. Pilar lives in Cork. Therefore, Pilar lives in Ireland.

6.5 Summary

This chapter considered propositional and predicate calculus. Propositional logic is the study of propositions, and a proposition is a statement that is either true or false. A formula in propositional calculus may contain several variables, and the truth or falsity of the individual variables, and the meanings of the logical connectives determines the truth or falsity of the logical formula.

A rich set of connectives is employed in propositional calculus to combine propositions and to build up the well-formed formulae of the calculus. This includes the conjunction of two propositions ($A \wedge B$), the disjunction of two propositions ($A \vee B$) and the implication of two propositions ($A \Rightarrow B$). These connectives allow compound propositions to be formed, and the truth of the compound propositions is determined from the truth-values of the constituent propositions and the rules associated with the logical connectives. The meaning of the logical connectives is given by truth tables.

Propositional calculus is both complete and consistent with all true propositions deducible in the calculus, and there is no formula A such that both A and $\neg A$ are deducible in the calculus.

An argument in propositional logic consists of a sequence of formulae that are the premises of the argument and a further formula that is the conclusion of the argument. One elementary way to see if the argument is valid is to produce a truth table to determine if the conclusion is true whenever all of the premises are true. Other ways are to use semantic tableaux or natural deduction.

Predicates are statements involving variables, and these statements become propositions once the variables are assigned values. Predicate calculus allows expressions such as all members of the domain have a particular property to be expressed formally: e.g. $(\forall x)Px$, or that there is at least one member that has a particular property: e.g. $(\exists x)Px$.

Predicate calculus may be employed to specify the requirements for a proposed system and to give the definition of a piecewise defined function. Semantic tableaux may be used for determining the validity of arguments in propositional or predicate logic, and its approach is to negate the conclusion of an argument and to show that this results in inconsistency with the premises of the argument.

7.1 Introduction

In this chapter, we consider some advanced topics in logic including fuzzy logic, temporal logic, intuitionist logic, undefined values, logic and AI and theorem provers. Fuzzy logic is an extension of classical logic that acts as a mathematical model for vagueness, and it handles the concept of partial truth where truth-values lie between completely true and completely false. Temporal logic is concerned with the expression of properties that have time dependencies, and it allows temporal properties about the past, present and future to be expressed.

Brouwer and others developed intuitionist logic as the logical foundation for intuitionism. This was a controversial theory on the foundations of mathematics, and it was based on a rejection of the law of the excluded middle, and an insistence that an existence proof must be constructive yielding the desired entity. Martin L of successfully applied intuitionism to type theory in the 1970s.

Partial functions arise naturally in computer science, and such functions may fail to be defined for one or more values in their domain. One approach to dealing with partial functions is to employ a precondition, which restricts the application of the function to values where it is defined. We consider three approaches to deal with undefined values, including the logic of partial functions; Dijkstra's approach with his *can* and *cor* operators; and Parnas's approach which preserves a classical two-valued logic.

We examine the contribution of logic to the AI field and give a brief introduction to work done by theorem provers in supporting proof.

7.2 Fuzzy Logic

Fuzzy logic is a branch of *many-valued logic* that allows inferences to be made when dealing with vagueness, and it can handle problems with imprecise or incomplete data. It differs from classical two-valued propositional logic, in that it is based on degrees of truth, rather than on the standard binary truth-values of “true or false” (1 or 0) of propositional logic. That is, while statements made in propositional logic are either true or false (1 or 0), the truth-value of a statement made in fuzzy logic is a value between 0 and 1. Its value expresses the extent to which the statement is true, with a value of 1 expressing absolute truth, and a value of 0 expressing absolute falsity.

Fuzzy logic uses *degrees of truth* as a mathematical model for vagueness, and this is useful since statements made in natural language are often vague and have a certain (rather than an absolute) degree of truth. It is an extension of classical logic to handle the concept of partial truth, where the truth-value lies between completely true and completely false. Lofti Zadeh developed fuzzy logic at Berkley in the 1960s, and it has been successfully applied to expert systems and other areas of Artificial Intelligence.

For example, consider the statement “John is tall”. If John is 6 ft, 4 in., then we would say that this is a true statement (with a truth-value of 1) since John is well above average height. However, if John is 5 feet, 9 in. tall (around average height), then this statement has a degree of truth, and this could be indicated by a fuzzy truth-valued of 0.6. Similarly, the statement that today is sunny may be assigned a truth-value of 1 if there are no clouds, 0.8 if there are a small number of clouds and 0 if it is raining all day.

Propositions in fuzzy logic may be combined together to form compound propositions. Suppose X and Y are propositions in fuzzy logic, then compound propositions may be formed from the conjunction, disjunction and implication operators. The usual definition in fuzzy logic of the truth-values of the compound propositions formed from X and Y is given by:

$$\begin{aligned}\text{Truth}(\neg X) &= 1 - \text{Truth}(X) \\ \text{Truth}(X \text{ and } Y) &= \min(\text{Truth}(X), \text{Truth}(Y)) \\ \text{Truth}(X \text{ or } Y) &= \max(\text{Truth}(X), \text{Truth}(Y)) \\ \text{Truth}(X \rightarrow Y) &= \text{Truth}(\neg X \text{ or } Y)\end{aligned}$$

There is another way in which the operators may be defined in terms of multiplication:

$$\begin{aligned}\text{Truth}(X \text{ and } Y) &= \text{Truth}(X) * \text{Truth}(Y) \\ \text{Truth}(X \text{ or } Y) &= 1 - (1 - \text{Truth}(X)) * (1 - \text{Truth}(Y)) \\ \text{Truth}(X \rightarrow Y) &= \max\{z | \text{Truth}(X) * z \leq \text{Truth}(Y)\} \text{ where } 0 \leq z \leq 1\end{aligned}$$

Under these definitions, fuzzy logic is an extension of classical two-valued logic, which preserves the usual meaning of the logical connectives of propositional logic when the fuzzy values are just $\{0,1\}$.

Fuzzy logic has been very useful in expert system and Artificial Intelligence applications. The first fuzzy logic controller was developed in England in the mid-1970s. It has been applied to the aerospace and automotive sectors, and also to the medical, robotics and transport sectors.

7.3 Temporal Logic

Temporal logic is concerned with the expression of properties that have time dependencies, and the various temporal logics express facts about the past, present and future. Temporal logic has been applied to specify temporal properties of natural language, Artificial Intelligence and the specification and verification of program and system behaviour. It provides a language to encode temporal knowledge in Artificial Intelligence applications, and it plays a useful role in the formal specification and verification of temporal properties (e.g. liveness and fairness) in safety critical systems.

The statements made in temporal logic can have a truth-value that varies over time. In other words, sometimes the statement is true and sometimes it is false, but it is never true or false at the same time. The two main types of temporal logics are *linear time logics* (reason about a single timeline) and *branching time logics* (reason about multiple timelines).

The roots of temporal logic lie in work done by Aristotle in the fourth century B.C., when he considered whether a truth-value should be given to a statement about a future event that may or may not occur. For example, what truth-value (if any) should be given to the statement that “*There will be a sea battle tomorrow*”. Aristotle argued against assigning a truth-value to such statements in the present time.

Newtonian mechanics assumes an absolute concept of time independent of space, and this viewpoint remained dominant until the development of the theory of relativity in the early twentieth century (when space–time became the dominant paradigm).

Arthur Prior began analysing and formalizing the truth-values of statements concerning future events in the 1950s, and he introduced tense logic (a temporal logic) in the early 1960s. Tense logic contains four modal operators (strong and weak) that express events in the future or in the past:

- P (It has at some time been the case that)
- F (It will be at some time be the case that)
- H (It has always been the case that)
- G (It will always be the case that)

The P and F operators are known as weak tense operators, while the H and G operators known as strong tense operators. The two pairs of operators are interdefinable via the equivalences:

$$P\phi \cong \neg H\neg\phi$$

$$H\phi, \cong \neg P\neg\phi$$

$$F\phi \cong \neg G\neg\phi$$

$$G\phi \cong \neg F\neg\phi$$

The set of formulae in Prior's temporal logic may be defined recursively, and they include the connectives used in classical logic (e.g. \neg , \wedge , \vee , \rightarrow , \leftrightarrow). We can express a property ϕ that is always true as $A\phi \cong H\phi \wedge \phi \wedge G\phi$ and a property that is sometimes true as $E\phi \cong P\phi \vee \phi \vee F\phi$. Various extensions of Prior's tense logic have been proposed to enhance its expressiveness. These include the binary *since* temporal operator " S ", and the binary *until* temporal operator " U ". For example, the meaning of $\phi S\psi$ is that ϕ has been true since a time when ψ was true.

Temporal logics are applicable to the specification of computer systems, as a specification may require *safety*, *fairness* and *liveness properties* to be expressed. For example, a fairness property may state that it will always be the case that a certain property will hold sometime in the future. The specification of temporal properties often involves the use of special temporal operators.

We discuss common temporal operators that are used, including an operator to express properties that will always be true; properties that will eventually be true; and a property that will be true in the next time instance. For example:

- $\square P$ P is always true
- $\bullet P$ P will be true sometime in the future
- $\circ P$ P is true in the next time instant (*discrete time*)

Linear temporal logic (LTL) was introduced by Pnueli in the late 1970s. This linear time logic is useful in expressing safety and liveness properties. Branching time logics assume a non-deterministic branching future for time (with a deterministic, linear past). Computation tree logic (CTL and CTL*) were introduced in the early 1980s by Clarke and Emerson [CM:81].

It is also possible to express temporal operations directly in classical mathematics, and the well-known computer scientist, Parnas, prefers this approach. He is critical of computer scientists for introducing unnecessary formalisms when classical mathematics already possesses the ability to do this. For example, the value of a function f at a time instance prior to the current time t is defined as:

$$Prior(f, t) = \lim_{\varepsilon \rightarrow 0} f(t - \varepsilon)$$

Temporal logic will be discussed again later in this book as part of model checking (in Chap. 14). For more detailed information on temporal logic, the reader is referred to the excellent article on temporal logic in [STL:15].

7.4 Intuitionist Logic

The controversial school of intuitionist mathematics was founded by the Dutch mathematician, L.E.J. Brouwer, who was a famous topologist, and well known for his fixpoint theorem in topology. This constructive approach to mathematics proved to be highly controversial, as its acceptance as a foundation of mathematics would have led to the rejection of many accepted theorems in classical mathematics (including his own fixed point theorem).

Brouwer was deeply interested in the foundations of mathematics, and the problems arising from the paradoxes of set theory. He was determined to provide a secure foundation for mathematics, and his view was that an existence theorem in mathematics has no validity, unless it is constructive and accompanied by a procedure to construct the object. He therefore rejected indirect proof and the law of the excluded middle ($P \vee \neg P$) or equivalently ($\neg\neg P \rightarrow P$), and he insisted on an explicit construction of the mathematical object.

The problem with the law of the excluded middle (LEM) arises in dealing with properties of infinite sets. For finite sets, one can decide if all elements of the set possess a certain property P by testing each one. However, this procedure is no longer possible for infinite sets. We may know that a certain element of the infinite set does not possess the property, or it may be the actual method of construction of the set allows us to prove that every element has the property. However, the application of the law of the excluded middle is invalid for infinite sets, as we cannot conclude from the situation where not all elements of an infinite set possess a property P that there exists at least one element which does not have the property P . In other words, the law of the excluded middle may only be applied in cases where the conclusion can be reached in a finite number of steps.

Consequently, if the Brouwer view of the world was accepted, then many of the classical theorems of mathematics (including his own well-known results in topology) could no longer be said to be true. His approach to the foundations of mathematics hardly made him popular with other contemporary mathematicians (the differences were so fundamental that it was more like a war between them), and intuitionism never became mainstream in mathematics. It led to deep and bitter divisions between Hilbert and Brouwer, with Hilbert accusing Brouwer (and Weyl) of trying to overthrow everything that did not suit them in mathematics and that intuitionism was treason to science. Hilbert argued that a suitable foundation for mathematics should aim to preserve most of mathematics. Brouwer described Hilbert's formalist program as a false theory that would produce nothing of mathematical value. For Brouwer, "to exist" is synonymous with "constructive existence", and constructive mathematics is

relevant to computer science, as a program may be viewed as the result obtained from a constructive proof of its specification.

Brouwer developed one of the more unusual logics that have been invented (intuitionist logic), in which many of the results of classical mathematics were no longer true. Intuitionist logic may be considered the logical basis of constructive mathematics, and formal systems for intuitionist propositional and predicate logic were developed by Heyting and others [Hey:66].

Consider a hypothetical mathematical property $P(x)$ of which there is no known proof (i.e. it is unknown whether $P(x)$ is true or false for arbitrary x where x ranges over the natural numbers). Therefore, the statement $\forall x (P(x) \vee \neg P(x))$ cannot be asserted with the present state of knowledge, as neither $P(x)$ or $\neg P(x)$ has been proved. That is, unproved statements in intuitionist logic are not given an intermediate truth-value, and they remain of an unknown truth-value until they have been either proved or disproved.

The intuitionist interpretation of the logical connectives is different from classical propositional logic. A sentence of the form $A \vee B$ asserts that either a proof of A or a proof of B has been constructed, and $A \vee B$ is not equivalent to $\neg(\neg A \wedge \neg B)$. Similarly, a proof of $A \wedge B$ is a pair whose first component is a proof of A , and whose second component is a proof of B . The statement $\forall x \neg P(x)$ is not equivalent to $\exists x P(x)$ in intuitionist logic.

Intuitionist logic was applied to type theory by Martin L of in the 1970s [Lof:84]. Intuitionist type theory is based on an analogy between propositions and types, where $A \wedge B$ is identified with $A \times B$, the Cartesian product of A and B . The elements in the set $A \times B$ are of the form (a, b) where $a \in A$ and $b \in B$. The expression $A \vee B$ is identified with $A + B$, the disjoint union of A and B . The elements in the set $A + B$ are got from tagging elements from A and B , and they are of the form $\text{inl}(a)$ for $a \in A$, and $\text{inr}(b)$ for $b \in B$. The left and right injections are denoted by inl and inr .

7.5 Undefined Values

Total functions $f: X \rightarrow Y$ are functions that are defined for every element in their domain, and total functions are widely used in mathematics. However, there are functions that are undefined for one or more elements in their domain, and one example is the function $y = 1/x$ which is undefined at $x = 0$.

Partial functions arise naturally in computer science, and such functions may fail to be defined for one or more values in their domain. One approach to dealing with partial functions is to employ a precondition, which restricts the application of the function to where it is defined. This makes it possible to define a new set (a proper subset of the domain of the function) for which the function is total over the new set.

Undefined terms often arise¹ and need to be dealt with. Consider, the example of the square root function \sqrt{x} taken from [Par: 93]. The domain of this function is the positive real numbers, and the following expression is undefined:

$$((x > 0) \wedge (y = \sqrt{x})) \vee ((x \leq 0) \wedge (y = \sqrt{-x}))$$

The reason this expression is undefined is since the usual rules for evaluating such an expression involve evaluating each subexpression, and then performing the Boolean operations. However, when $x < 0$, the subexpression $y = \sqrt{x}$ is undefined, whereas when $x > 0$ the subexpression $y = \sqrt{-x}$ is undefined. Clearly, it is desirable that such expressions be handled and that for the example above, the expression would evaluate to be true.

Classical two-valued logic does not handle this situation adequately, and there have been several proposals to deal with undefined values. Dijkstra’s approach is to use the *cand* and *cor* operators in which the value of the left-hand operand determines whether the right-hand operand expression is evaluated or not. Jones’s logic of partial functions [Jon:86] uses a three-valued logic², and Parnas’s³ approach is an extension to the predicate calculus to deal with partial functions that preserve the two-valued logic.

7.5.1 Logic of Partial Functions

Jones [Jon:86] has proposed the logic of partial functions (LPFs) as an approach to deal with terms that may be undefined. This is a three-valued logic, and a logical term may be true, false, or undefined (denoted \perp). The definition of the truth functional operators used in classical two-valued logic is extended to three-valued logic. The truth tables for conjunction and disjunction are defined in Fig. 7.1.

The conjunction of P and Q is true when both P and Q are true; false if one of P or Q is false; and undefined, otherwise. The operation is commutative. The disjunction of P and Q ($P \vee Q$) is true if one of P or Q is true; false if both P and

		\wedge Q						\vee Q					
		T	F	\perp	P \wedge Q			T	F	\perp	P \vee Q		
P	T	T	F	\perp	T	T	T	T	T	T	T	T	T
	F	F	F	F	F	F	F	F	F	F	\perp	\perp	\perp
	\perp	\perp	F	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp	\perp

Fig. 7.1 Conjunction and disjunction operators for LPF

¹It is best to avoid undefinedness by taking care with the definitions of terms and expressions.
²The above expression would evaluate to true under Jones three-valued logic of partial functions.
³The above expression evaluates to true for Parnas logic (a two-valued logic).

\rightarrow	Q	T	F	\perp	\leftrightarrow	Q	T	F	\perp
P		$P \rightarrow Q$			P		$P \leftrightarrow Q$		
T		T	F	\perp	T		T	F	\perp
F		T	T	T	F		F	T	\perp
\perp		T	\perp	\perp	\perp		\perp	\perp	\perp

Fig. 7.2 Implication and equivalence operators for LPF

A	$\neg A$
T	F
F	T
\perp	\perp

Fig. 7.3 Negation for LPF

Q are false; and undefined, otherwise. The implication operation ($P \rightarrow Q$) is true when P is false or when Q is true; false when P is true and Q is false; and undefined, otherwise. The equivalence operation ($P \leftrightarrow Q$) is true when both P and Q are true or false; it is false when P is true and Q is false (and vice versa); and it is undefined, otherwise (Fig. 7.2).

The not operator (\neg) is a unary operator such $\neg A$ is true when A is false, false when A is true, and undefined when A is undefined (Fig. 7.3).

The result of an operation may be known immediately after knowing the value of one of the operands (e.g. disjunction is true if P is true irrespective of the value of Q). The law of the excluded middle; that is, $A \vee \neg A$ does not hold in the three-valued logic, and Jones [Jon:86] argues that this is reasonable as one would not expect the following to be true:

$$(\perp / 0 = 1) \vee (\perp / 0 \neq 1)$$

There are other well-known laws that fail to hold such as:

- (i) $E \Rightarrow E$.
- (ii) Deduction theorem $E_1 \vdash E_2$ does not justify $\vdash E_1 \Rightarrow E_2$ unless it is known that E_1 is defined.

Many of the tautologies of standard logic also fail to hold.

Table 7.1 Examples of Parnas evaluation of undefinedness

Expression	$x < 0$	$x \geq 0$
$y = \sqrt{x}$	<i>False</i>	<i>True if $y = \sqrt{x}$, False otherwise</i>
$y = 1/0$	<i>False</i>	<i>False</i>
$y = x^2 + \sqrt{x}$	<i>False</i>	<i>True if $y = x^2 + \sqrt{x}$, False otherwise</i>

Table 7.2 Example of undefinedness in array

Expression	$i \in \{1 \dots N\}$	$i \notin \{1 \dots N\}$
$B[i] = x$	<i>True if $B[i] = x$</i>	<i>False</i>
$\exists i, B[i] = x$	<i>True if $B[i] = x$ for some i, False otherwise</i>	<i>False</i>

7.5.2 Parnas Logic

Parnas’s approach to logic is based on classical two-valued logic, and his philosophy is that truth-values should be true or false only⁴ and that there is no third logical value. It is an extension to predicate calculus to deal with partial functions. The evaluation of a logical expression yields the value “true” or “false” irrespective of the assignment of values to the variables in the expression. This allows the expression: $(y = \sqrt{x}) \vee (y = \sqrt{-x})$ that is undefined in classical logic to yield the value true.

The advantages of his approach are those no new symbols are introduced into the logic, and the logical connectives retain their traditional meaning. This makes it easier for engineers and computer scientists to understand, as it is closer to their intuitive understanding of logic.

The meaning of predicate expressions is given by first defining the meaning of the primitive expressions. These are then used as the building blocks for predicate expressions. The evaluation of a primitive expression $R_j(V)$ (where V is a comma separated set of terms with some elements of V involving the application of partial functions) is false if the value of an argument of a function used in one of the terms of V is not in the domain of that function.⁵ The following examples (Tables 7.1 and 7.2) should make this clearer.

These primitive expressions are used to build the predicate expressions, and the standard logical connectives are used to yield truth-values for the predicate expression. Parnas logic is defined in detail in [Par:93].

Parnas logic may be seen more clearly by considering a tabular expressions example from [Par:93]. Figure 7.4 specifies the behaviour of a program that searches the array B for the value x . It describes the properties of the values of j' and $present'$. There are two cases to consider:

⁴It seems strange to assign the value false to the primitive predicate calculus expression $y = 1/0$.

⁵The approach avoids the undefined logical value (\perp) and preserves the two-valued logic.

H_1	j'	$(\exists i, B[i]=x)$	$\neg(\exists i, B[i]=x)$	H_2
	present' =	$B[j]=x$	<u>true</u>	
		true	false	G

Fig. 7.4 Finding index in array

1. There is an element in the array with the value of x .
2. There is no such element in the array with the value of x .

Clearly, from the example above the predicate expressions $\exists i, B[i] = x$ and $\neg(\exists i, B[i] = x)$ are defined. One disadvantage of the Parnas's approach is that some common relational operators (e.g. $>$, \geq , \leq and $<$) are not primitive in the logic. However, these relational operators are then constructed from primitive operators. Further, the axiom of reflection does not hold in the logic.

7.5.3 Dijkstra and Undefinedness

The *and* and *cor* operators were introduced by Dijkstra to deal with undefined values. They are non-commutative operators and allow the evaluation of predicates that contain undefined values. Consider the following expression:

$$y = 0 \vee \left(\frac{x}{y} = 2 \right)$$

Then, this expression is undefined when $y = 0$ as x/y is undefined, since the logical disjunction operation is not defined when one of its operands is undefined. However, there is a case for giving meaning to such an expression when $y = 0$, since in that case the first operand of the logical or operation is true. Further, the logical *disjunction* operation is defined to be true if either of its operands is true. This motivates the introduction of the *and* and *cor* operators. These operators are associative, and their truth tables are defined in Tables 7.3 and 7.4:

The order of the evaluation of the operands for the *and* operation is to *evaluate the first operand*; if the first operand is true, then the result of the operation is the second operand; otherwise, the result is false. The expression $a \text{ and } b$ is equivalent to:

$$a \text{ and } b \cong \text{if } a \text{ then } b \text{ else } F$$

The order of the evaluation of the operands for the *cor* operation is to evaluate the first operand. If the first operand is true, then the result of the operation is true; otherwise, the result of the operation is the second operand. The expression $a \text{ cor } b$ is equivalent to:

Table 7.3 $a \text{ cand } b$

a	b	$a \text{ cand } b$
T	T	T
T	F	F
T	U	U
F	T	F
F	F	F
F	U	F
U	T	U
U	F	U
U	U	U

Table 7.4 $a \text{ cor } b$

a	b	$a \text{ cor } b$
T	T	T
T	F	T
T	U	T
F	T	T
F	F	F
F	U	U
U	T	U
U	F	U
U	U	U

$$a \text{ cor } b \cong \text{if } a \text{ then } T \text{ else } b$$

The **cand** and **cor** operators satisfy the following laws:

- *Associativity*

The **cand** and **cor** operators are associative.

$$(A \text{ cand } B) \text{ cand } C = A \text{ cand } (B \text{ cand } C)$$

$$(A \text{ cor } B) \text{ cor } C = A \text{ cor } (B \text{ cor } C)$$

- *Distributivity*

The **cand** operator distributes over the **cor** operator and vice versa.

$$A \text{ cand } (B \text{ cor } C) = (A \text{ cand } B) \text{ cor } (A \text{ cand } C)$$

$$A \text{ cor } (B \wedge C) = (A \text{ cor } B) \text{ cand } (A \text{ cor } C)$$

De Morgan's law enables logical expressions to be simplified.

$$\neg(A \text{ \textbf{cand}} B) = \neg A \text{ \textbf{cor}} \neg B$$

$$\neg(A \text{ \textbf{cor}} B) = \neg A \text{ \textbf{cand}} \neg B$$

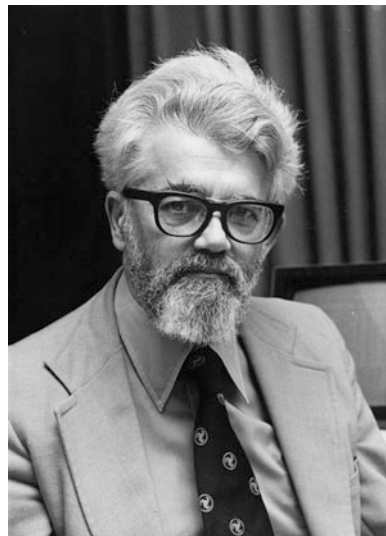
7.6 Logic and AI

The long-term goal of Artificial Intelligence is to create a thinking machine that is intelligent, has consciousness, has the ability to learn, has free will and is ethical. Artificial Intelligence is a young field, and John McCarthy and others coined the term in 1956. Alan Turing devised the Turing Test in the early 1950s as a way to determine whether a machine was conscious and intelligent. Turing believed that machines would eventually be developed that would stand a good chance of passing the "Turing Test".

There are deep philosophical problems in Artificial Intelligence, and some researchers believe that its goals are impossible or incoherent. Even if Artificial Intelligence is possible, there are moral issues to consider such as the exploitation of artificial machines by humans and whether it is ethical to do this. Weizenbaum argues that AI is a threat to human dignity and that AI should not replace humans in positions that require respect and care.

John McCarthy (Fig. 7.5) has long advocated the use of logic in AI, and mathematical logic has been used in the AI field to formalize knowledge, and in guiding the design of mechanized reasoning systems. Logic has been used as an analytic tool, as a knowledge representation formalism and as a programming language.

Fig. 7.5 John McCarthy.
Courtesy of John McCarthy



McCarthy's long-term goal was to formalize common-sense reasoning, i.e. the normal reasoning that is employed in problem solving and dealing with normal events in the real world. McCarthy [Mc:59] argues that it is reasonable for logic to play a key role in the formalization of common-sense knowledge, and this includes the formalization of basic facts about actions and their effects; facts about beliefs and desires; and facts about knowledge and how it is obtained. His approach allows common-sense problems to be solved by logical reasoning.

Its formalization requires sufficient understanding of the common-sense world, and often the relevant facts to solve a particular problem are unknown. It may be that knowledge thought relevant may be irrelevant and vice versa. A computer may have millions of facts stored in its memory, and the (challenging) problem is how to determine which of these should be chosen from its memory to serve as premises in logical deduction.

McCarthy's influential 1959 paper discusses various common-sense problems such as getting home from the airport. Mathematical logic is the standard approach to express premises, and it includes rules of inferences those are used to deduce valid conclusions from a set of premises. Its rigorous deductive reasoning shows how new formulae may be logically deduced from a set or premises.

McCarthy's approach to programs with common sense has been criticized by Bar-Hillel and others on the grounds that common sense is fairly elusive, and the difficulty that a machine would have in determining which facts are relevant to a particular deduction from its known set of facts. However, McCarthy's approach has showed how logical techniques can contribute to the solution of specific AI problems.

Logic-programming languages describe what is to be done, rather than how it should be done. These languages are concerned with the statement of the problem to be solved, rather than how the problem will be solved. Mathematical logic is used as a tool in the statement of the problem definition, and logic is useful in developing a body of knowledge (or theory). Further, it allows rigorous mathematical deduction of further truths from the existing set of truths. The theory is built up from a small set of axioms or postulates, and the rules of inference are used to derive further truths logically.

The objective of logic programming is to employ mathematical logic to assist with computer programming. Many problems are naturally expressed as a theory, and the statement of a problem to be solved is often equivalent to determining if a new hypothesis is consistent with an existing theory. Logic provides a rigorous way to determine this, as it includes a rigorous process for conducting proof.

Computation in logic programming is essentially logical deduction, and logic-programming languages use first-order⁶ predicate calculus. Theorem proving is employed to derive a desired truth from an initial set of axioms. These proofs are constructive⁷ in that an actual object that satisfies the constraints is produced rather than a pure existence theorem. Logic programming specifies the objects, the relationships between them and the constraints that must be satisfied for the problem.

- The set of objects involved in the computation;
- The relationships that hold between the objects;
- The constraints of the particular problem.

The language interpreter decides how to satisfy the particular constraints. Artificial Intelligence influenced the development of logic programming, and John McCarthy⁸ demonstrated that mathematical logic could be used for expressing knowledge. The first logic-programming language was Planner developed by Carl Hewitt at MIT in 1969. It uses a procedural approach for knowledge representation rather than McCarthy's declarative approach.

The best-known logic-programming language is Prolog, which was developed in the early 1970s by Alain Colmerauer and Robert Kowalski. It stands for *programming in logic*. It is a goal-oriented language that is based on predicate logic. Prolog became an ISO standard in 1995. The language attempts to solve a goal by tackling the subgoals that the goal consists of:

$$\text{goal} : - \text{subgoal}_1, \dots, \text{subgoal}_n.$$

That is, in order to prove a particular goal, it is sufficient to prove subgoal₁ through subgoal_n. Each line of a Prolog program consists of a rule or a fact, and the language specifies what exists rather than how. The following program fragment has one rule and two facts:

$$\begin{aligned} \text{grandmother}(G, S) : - & \text{parent}(P, S), \text{mother}(G, P). \\ & \text{mother}(\text{sarah}, \text{isaac}). \\ & \text{parent}(\text{isaac}, \text{jacob}). \end{aligned}$$

⁶First-order logic allows quantification over objects but not functions or relations. Higher-order logics allow quantification of functions and relations.

⁷For example, the statement $\exists x$ such that $x = \sqrt{4}$ states that there is an x such that x is the square root of 4, and the constructive existence yields that the answer is that $x = 2$ or $x = -2$ i.e. constructive existence provides more the truth of the statement of existence, and an actual object satisfying the existence criteria is explicitly produced.

⁸John McCarthy received the Turing Award in 1971 for his contributions to Artificial Intelligence. He also developed the programming language LISP.

The first line in the program fragment is a rule that states that G is the grandmother of S if there is a parent P of S and G is the mother of P. The next two statements are facts stating that isaac is a parent of jacob and that sarah is the mother of isaac. A particular goal clause is true if all of its subclauses are true:

$$\text{goalclause}(V_g) : - \text{clause}_1(V_1), \dots, \text{clause}_m(V_m)$$

A Horn clause consists of a goal clause and a set of clauses that must be proven separately. Prolog finds solutions by *unification*, i.e. by binding a variable to a value. For an implication to succeed, all goal variables V_g on the left side of :- must find a solution by binding variables from the clauses which are activated on the right side. When all clauses are examined and all variables in V_g are bound, the goal succeeds. But if a variable cannot be bound for a given clause, then that clause fails. Following the failure, Prolog *backtracks*, and this involves going back to the left to previous clauses to continue trying to unify with alternative bindings. Backtracking gives Prolog the ability to find multiple solutions to a given query or goal.

Logic-programming languages generally use a simple searching strategy to consider alternatives:

- If a goal succeeds and there are more goals to achieve, then remember any untried alternatives and go on to the next goal.
- If a goal is achieved and there are no more goals to achieve, then stop with success.
- If a goal fails and there are alternative ways to solve it, then try the next one.
- If a goal fails and there are no alternate ways to solve it, and there is a previous goal, then go back to the previous goal.
- If a goal fails and there are no alternate ways to solve it, and no previous goal, then stop with failure.

Constraint programming is a programming paradigm where relations between variables can be stated in the form of constraints. Constraints specify the properties of the solution and differ from the imperative programming languages in that they do not specify the sequence of steps to execute.

7.7 Theorem Provers for Logic

The word “*proof*” is generally interpreted as facts or evidence that support a particular proposition or belief, and such proofs are conducted in natural language. The proof of a theorem in mathematics requires additional rigour, and such proofs consist of a mixture of natural language and mathematical argument. It is common to skip over the trivial steps in a mathematical proof, and independent mathematicians conduct peer reviews to provide additional confidence in the correctness of the proof, and to ensure that no unwarranted assumptions or errors in reasoning

have been made. Proofs conducted in logic are extremely rigorous with every step in the proof explicit.⁹

Herbert Simon and Alan Newell developed the first theorem prover with their work on a program called “Logic Theorist” or “LT” [NeS:56]. This program could independently provide proofs of various theorems in Russell’s and Whitehead’s *Principia Mathematica*¹⁰[RuW:10]. Russell and Whitehead had attempted to derive all mathematics from axioms and the inference rules of logic, and the LT program conducted proof from a small set of propositional axioms and deduction rules. The LT program succeeded in proving 38 of the 52 theorems in Chap. 2 of *Principia Mathematica*. Its approach was to start with the theorem to be proved, and to then search for relevant axioms and operators to prove the theorem.

LT was demonstrated at the Dartmouth conference in 1956 (the conference that led to the birth of the Artificial Intelligence field), and it showed that computers had the ability to encode knowledge and information, and to perform intelligent operations such as solving theorems in mathematics. The heuristic approach of the LT program tried to emulate human mathematicians, but could not guarantee that a proof could be found for every valid theorem.

The proof of theorems in formal verification of computer system often involves several million formulae, and manual proof is error prone. There are several tools available to support theorem proving, and these include the Boyer-Moore theorem prover (also known as NQTHM); the Isabelle theorem prover; and the HOL system.

B.S. Boyer and J.S. Moore developed the Boyer-Moore theorem prover in the early 1970s [BoM:79]. It has been improved since then, and it is currently known as NQTHM (it has been superseded by ACL2 available from the University of Texas). It has been effective in proving well-known theorems such as Gödel’s incompleteness theorem, the insolvability of the Halting problem, a formalization of the Motorola MC 68020 Microprocessor and many more.

Computational Logic Inc. was a company founded by Boyer and Moore in 1983 to share the benefits of a formal approach to software development with the wider computing community. It was based in Austin, Texas, and provided services in the mathematical modelling of hardware and software systems. This involved the use of mathematics and logic to formally specify microprocessors and other systems. The use of its theorem prover was to formally verify that the implementation meets its specification, i.e. to prove that the microprocessor or other system satisfies its specification.

Isabelle is a theorem-proving environment developed at Cambridge University by Larry Paulson and Tobias Nipkow of the Technical University of Munich. It allows mathematical formulae to be expressed in a formal language and provides

⁹Perhaps a good analogy might be that a mathematical proof is like a program written in a high-level language such as C, whereas a formal proof in logic is like a program written in assembly language.

¹⁰Russell is said to have remarked that he was delighted to see that the *Principia Mathematica* could be done by machine and that if he and Whitehead had known this in advance that they would not have wasted 10 years doing this work by hand in the early twentieth century.

tools for proving those formulae. The main application is the formalization of mathematical proofs, and proving the correctness of computer hardware or software with respect to its specification, and proving properties of computer languages and protocols.

Isabelle is a generic theorem prover in the sense that it has the capacity to accept a variety of formal calculi, whereas most other theorem provers are specific to a specific formal calculus. Isabelle is available under an open-source licence.

The HOL system is an environment for interactive theorem proving in a higher-order logic. The HOL system has been applied to the formalization of mathematics and the verification of hardware. It was originally developed at Cambridge University in the UK, in the early 1980s, and HOL 4 is the latest version and is an open-source project. It is used by academia and industry.

There is a steep learning curve with theorem provers above, and it generally takes a couple of months for users to become familiar with them. However, automated theorem proving has become a useful tool in the verification of integrated circuit design. Several semiconductor companies use automated theorem proving to demonstrate the correctness of division and other operators on their processors. The nature of proof and theorem provers is discussed in Chap. 15.

7.8 Review Questions

1. What is fuzzy logic?
2. What is intuitionist logic and how is it different from classical logic?
3. Discuss the problem of undefinedness and the advantages and disadvantages of three-valued logics. Describe the approaches of Parnas, Dijkstra and Jones.
4. What is temporal logic?
5. Show how temporal operators may be expressed in classical mathematics.
6. Investigate the Isabelle (or another) theorem-proving environment and determine the extent to which it may assist with proof.
7. Discuss the applications of logic to AI.

7.9 Summary

We discussed some advanced topics in logic in this chapter, including fuzzy logic, temporal logic, intuitionist logic, undefined values, logic and AI and theorem provers. Fuzzy logic is an extension of classical logic that acts as a mathematical model for vagueness, whereas temporal logic is concerned with the expression of properties that have time dependencies

Intuitionism was a controversial school of mathematics that aimed to provide a solid foundation for mathematics. Its adherents rejected the law of the excluded middle and insisted that for an entity to exist that there must be a constructive proof of its existence. Martin L of applied intuitionistic logic to type theory in the 1970s.

Partial functions arise naturally in computer science, and such functions may fail to be defined for one or more values in their domain. There are a number of approaches to deal with undefined values, including Jones's logic of partial functions; Dijkstra's approach with his *cand* and *cor* operators; and Parnas's approach which preserves a classical two-valued logic.

We discussed temporal logic and its applications to the safety critical field, including the specification of properties with time dependencies. We discussed the application of logic to the AI field, and logic has been used to formalize knowledge in an AI systems. Finally, we discussed some of the existing theorem provers, and their applications in providing a rigorous proof of a theorem, and in avoiding errors or jumps in reasoning.

8.1 Introduction

Z is a formal specification language based on Zermelo set theory. It was developed at the Programming Research Group at Oxford University in the early 1980s [Dil:90] and became an ISO standard in 2002. Z specifications are mathematical and employ a classical two-valued logic. The use of mathematics ensures precision and allows inconsistencies and gaps in the specification to be identified. Theorem provers may be employed to demonstrate that the software implementation meets its specification.

Z is a “*model-oriented*” approach with an explicit model of the state of an abstract machine given, and operations are defined in terms of this state. Its mathematical notation is used for formal specification, and the schema calculus is used to structure the specifications. The schema calculus is visually striking, and consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schemas may be used as building blocks and combined with other schemas. The simple schema (Fig. 8.1) is the specification of the positive square root of a real number.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make Z specifications highly readable, as each individual schema is small in size and self-contained. Exception handling is addressed by defining schemas for the exception cases. These are then combined with the original operation schema. Mathematical data types are used to model the data in a system, and these data types obey mathematical laws. These laws enable simplification of expressions and are useful with proofs.

Operations are defined in a precondition/postcondition style. A precondition must be true before the operation is executed, and the postcondition must be true after the operation has executed. The precondition is implicitly defined within the operation. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The system

$$\left\{ \begin{array}{l} \text{-SqRoot-----} \\ \text{num?}, \text{root!} : \mathbb{R} \\ \text{-----} \\ \text{num?} \geq 0 \\ \text{root!}^2 = \text{num?} \\ \text{root!} \geq 0 \\ \text{-----} \end{array} \right.$$

Fig. 8.1 Specification of positive square root

invariant is a property of the system that must be true at all times. The initial state itself is, of course, required to satisfy the system invariant.

The precondition for the specification of the square root function above is that $\text{num?} \geq 0$; i.e. the function *SqRoot* may be applied to positive real numbers only. The postcondition for the square root function is $\text{root!}^2 = \text{num?}$ and $\text{root!} \geq 0$. That is, the square root of a number is positive and its square gives the number. Postconditions employ a logical predicate which relates the prestate to the poststate, with the poststate of a variable being distinguished by priming the variable, e.g. v' .

Z is a typed language and whenever a variable is introduced its type must be given. A type is simply a collection of objects, and there are several standard types in Z. These include the natural numbers \mathbb{N} , the integers \mathbb{Z} and the real numbers \mathbb{R} . The declaration of a variable x of type X is written $x: X$. It is also possible to create your own types in Z.

Various conventions are employed within Z specification, for example $v?$ indicates that v is an input variable; $v!$ indicates that v is an output variable. The variable num? is an input variable and root! is an output variable for the square root example above. The notation Ξ in a schema indicates that the operation *Op* does not affect the state, whereas the notation Δ in the schema indicates that *Op* is an operation that affects the state.

Many of the data types employed in Z have no counterpart in standard programming languages. It is therefore important to identify and describe the concrete data structures that ultimately will represent the abstract mathematical structures. As the concrete structures may differ from the abstract, the operations on the abstract data structures may need to be refined to yield operations on the concrete data that yield equivalent results. For simple systems, direct refinement (i.e. one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement¹ is employed, where a sequence of increasingly concrete specifications are produced to yield the executable specification. There is a calculus for combining schemas to make larger specifications, and this is discussed later in the chapter.

Example 8.1 The following is a Z specification to borrow a book from a library system. The library is made up of books that are on the shelf; books that are borrowed and books that are missing (Fig. 8.2). The specification models a library

¹Step-wise refinement involves producing a sequence of increasingly more concrete specifications until eventually the executable code is produced. Each refinement step has associated proof obligations to prove that it is valid.

$$\begin{array}{|l}
 \hline
 \text{-Library-----} \\
 \text{on-shelf, missing, borrowed} : \mathbb{P} \text{ Bkd-Id} \\
 \hline
 \text{on-shelf} \cap \text{missing} = \emptyset \\
 \text{on-shelf} \cap \text{borrowed} = \emptyset \\
 \text{borrowed} \cap \text{missing} = \emptyset \\
 \hline
 \end{array}$$

Fig. 8.2 Specification of a library system

$$\begin{array}{|l}
 \hline
 \text{-Borrow-----} \\
 \Delta \text{ Library} \\
 b? : \text{Bkd-Id} \\
 \hline
 b? \in \text{on-shelf} \\
 \text{on-shelf}' = \text{on-shelf} \setminus \{b?\} \\
 \text{borrowed}' = \text{borrowed} \cup \{b?\} \\
 \hline
 \end{array}$$

Fig. 8.3 Specification of borrow operation

with sets representing books on the shelf, on loan or missing. These are three mutually disjoint subsets of the set of books *Bkd-Id*.

The system state is defined in the *Library* schema below, and operations such as *Borrow* and *Return* affect the state. The *Borrow* operation is specified (Fig. 8.3).

The notation $\mathbb{P}Bkd-Id$ is used to represent the power set of *Bkd-Id* (i.e. the set of all subsets of *Bkd-Id*). The disjointness condition for the library is expressed by the requirement that the pair-wise intersection of the subsets *on-shelf*, *borrowed*, *missing* is the empty set.

The precondition for the *Borrow* operation is that the book must be available on the shelf to borrow. The postcondition is that the borrowed book is added to the set of borrowed books and is removed from the books on the shelf.

Z has been successfully applied in industry including the CICS project at IBM Hursley in the UK.² Next, we describe key parts of Z including sets, relations, functions, sequences and bags.

8.2 Sets

Sets were discussed in Chap. 4 and this section focuses on their use in Z. Sets may be enumerated by listing all of their elements. Thus, the set of all even natural numbers less than or equal to 10 is:

²This project claimed a 9% increase in productivity attributed to the use of formal methods.

$$\{2, 4, 6, 8, 10\}.$$

Sets may be created from other sets using set comprehension, i.e. stating the properties that its members must satisfy. For example, the set of even natural numbers less than 10 is given by set comprehension as:

$$\{n : \mathbb{N} \mid n \neq 0 \wedge n < 10 \wedge n \bmod 2 = 0 \bullet n\}$$

There are three main parts to the set comprehension above. The first part is the signature of the set and this is given by $n : \mathbb{N}$ above. The first part is separated from the second part by a vertical line. The second part is given by a predicate, and for this example the predicate is $n \neq 0 \wedge n < 10 \wedge n \bmod 2 = 0$. The second part is separated from the third part by a bullet. The third part is a term, and for this example it is simply n . The term is often a more complex expression: e.g. $\log(n^2)$.

In mathematics, there is just one empty set. However, since Z is a typed set theory, there is an empty set for each type of set. Hence, there are an infinite number of empty sets in Z. The empty set is written as $\emptyset [X]$ where X is the type of the empty set. In practice, X is omitted when the type is clear.

Various operations on sets such as union, intersection, set difference and symmetric difference are employed in Z. The power set of a set X is the set of all subsets of X and is denoted by $\mathbb{P}X$. The set of non-empty subsets of X is denoted by \mathbb{P}_1X where

$$\mathbb{P}_1X == \{U : \mathbb{P}X \mid U \neq \emptyset[X]\}.$$

A finite set of elements of type X (denoted by $F X$) is a subset of X that cannot be put into a one to one correspondence with a proper subset of itself. This is defined formally as:

$$F X == \{U : \mathbb{P} X \mid \neg \exists V : \mathbb{P} U \bullet V \neq U \wedge (\exists f : V \xrightarrow{\text{inj}} U)\}$$

The expression $f : V \xrightarrow{\text{inj}} U$ denotes that f is a bijection from U to V and injective, surjective and bijective functions were discussed in Chap. 4.

The fact that Z is a typed language means that whenever a variable is introduced (e.g. in quantification with \forall and \exists) it is first declared. For example, $\forall j : J \bullet P \Rightarrow Q$. There is also the unique existential quantifier $\exists_1 j : J \mid P$ which states that there is exactly one j of type J that has property P.

8.3 Relations

Relations are used extensively in Z and were discussed in Chap. 4. A relation R between X and Y is any subset of the Cartesian product of X and Y; i.e. $R \subseteq (X \times Y)$, and a relation in Z is denoted by $R : X \leftrightarrow Y$. The notation $x \mapsto y$ indicates that the pair $(x, y) \in R$.

Consider, the relation $home_owner: Person \leftrightarrow Home$ that exists between people and their homes. An entry $daphne \mapsto mandalay \in home_owner$ if $daphne$ is the owner of $mandalay$. It is possible for a person to own more than one home:

$$\begin{aligned} rebecca &\mapsto nirvana \in home_owner \\ rebecca &\mapsto tivoli \in home_owner. \end{aligned}$$

It is possible for two people to share ownership of a home:

$$\begin{aligned} rebecca &\mapsto nirvana \in home_owner \\ lawrence &\mapsto nirvana \in home_owner. \end{aligned}$$

There may be some people who do not own a home, and there is no entry for these people in the relation $home_owner$. The type $Person$ includes every possible person, and the type $Home$ includes every possible home. The domain of the relation $home_owner$ is given by:

$$x \in \text{dom } home_owner \Leftrightarrow \exists h : Home \bullet x \mapsto h \in home_owner.$$

The range of the relation $home_owner$ is given by:

$$h \in \text{ran } home_owner \Leftrightarrow \exists x : Person \bullet x \mapsto h \in home_owner.$$

The composition of two relations $home_owner: Person \leftrightarrow Home$ and $home_value: Home \leftrightarrow Value$ yields the relation $owner_wealth: Person \leftrightarrow Value$ and is given by the relational composition $home_owner; home_value$ where:

$$\begin{aligned} p \mapsto v \in home_owner; home_value &\Leftrightarrow \\ (\exists h : Home \bullet p \mapsto h \in home_owner \wedge h \mapsto v \in home_value). \end{aligned}$$

The relational composition may also be expressed as:

$$owner_wealth = home_value \circ home_owner.$$

The union of two relations often arises in practice. Suppose a new entry $aisling \mapsto muckross$ is to be added. Then this is given by

$$home_owner' = home_owner \cup \{aisling \mapsto muckross\}.$$

Suppose that we are interested in knowing all females who are house owners. Then we restrict the relation $home_owner$ so that the first element of all ordered pairs has to be female. Consider $female: \mathbb{P} Person$ with $\{aisling, rebecca\} \subseteq female$.

$$\begin{aligned} \text{home_owner} &= \{ \text{aisling} \mapsto \text{muckcross}, \text{rebecca} \mapsto \text{nirvana}, \text{lawrence} \mapsto \text{nirvana} \} \\ \text{female} \triangleleft \text{home_owner} &= \{ \text{aisling} \mapsto \text{muckcross}, \text{rebecca} \mapsto \text{nirvana} \}. \end{aligned}$$

That is, $\text{female} \triangleleft \text{home_owner}$ is a relation that is a subset of home_owner , and the first element of each ordered pair in the relation is female. The operation \triangleleft is termed domain restriction and its fundamental property is:

$$x \mapsto y \in U \triangleleft R \Leftrightarrow (x \in U \wedge x \mapsto y \in R),$$

where $R: X \leftrightarrow Y$ and $U: \mathbb{P} X$.

There is also a domain anti-restriction (subtraction) operation and its fundamental property is:

$$x \mapsto y \in U \triangleleft R \Leftrightarrow (x \notin U \wedge x \mapsto y \in R)$$

where $R: X \leftrightarrow Y$ and $U: \mathbb{P} X$.

There are also range restriction (the \triangleright operator) and the range anti-restriction operator (the \triangleright operator). These are discussed in [Dil:90].

8.4 Functions

A function [Dil:90] is an association between objects of some type X and objects of another type Y such that given an object of type X , there exists only one object in Y associated with that object. A function is a set of ordered pairs where the first element of the ordered pair has at most one element associated with it. A function is therefore a special type of relation, and a function may be *total* or *partial*.

A total function has exactly one element in Y associated with each element of X , whereas a partial function has at most one element of Y associated with each element of X (there may be elements of X that have no element of Y associated with them).

A partial function from X to Y ($f: X \rightarrow Y$) is a relation $f: X \leftrightarrow Y$ such that:

$$\forall x : X; y, z : Y \bullet (x \mapsto y \in f \wedge x \mapsto z \in f \Rightarrow y = z).$$

The association between x and y is denoted by $f(x) = y$, and this indicates that the value of the partial function f at x is y . A total function from X to Y (denoted $f: X \rightarrow Y$) is a partial function such that every element in X is associated with some value of Y .

$$f : X \rightarrow Y \Leftrightarrow f : X \leftrightarrow Y \wedge \text{dom} f = X.$$

Clearly, every total function is a partial function but not vice versa.

One operation that arises quite frequently in specifications is the function override operation. Consider the following specification of a temperature map:

$$\begin{array}{l} \hline \text{---TempMap---} \\ \text{CityList} : \mathbb{P}\text{City} \\ \text{temp} : \text{City} \rightarrow Z \\ \hline \text{dom temp} = \text{CityList} \\ \hline \end{array}$$

Suppose the temperature map is given by $\text{temp} = \{\text{Cork} \mapsto 17, \text{Dublin} \mapsto 19, \text{London} \mapsto 15\}$. Then consider the problem of updating the temperature map if a new temperature reading is made in Cork: e.g. $\{\text{Cork} \mapsto 18\}$. Then the new temperature chart is obtained from the old temperature chart by function override to yield $\{\text{Cork} \mapsto 18, \text{Dublin} \mapsto 19, \text{London} \mapsto 15\}$. This is written as:

$$\text{temp}' = \text{temp} \oplus \{\text{Cork} \mapsto 18\}.$$

The function override operation combines two functions of the same type to give a new function of the same type. The effect of the override operation is that the entry $\{\text{Cork} \mapsto 17\}$ is removed from the temperature chart and replaced with the entry $\{\text{Cork} \mapsto 18\}$.

Suppose $f, g: X \rightarrow Y$ are partial functions then $f \oplus g$ is defined and indicates that f is overridden by g . It is defined as follows:

$$\begin{aligned} (f \oplus g)(x) &= g(x) \text{ where } x \in \text{dom } g \\ (f \oplus g)(x) &= f(x) \text{ where } x \notin \text{dom } g \wedge x \in \text{dom } f. \end{aligned}$$

This may also be expressed (using domain anti-restriction) as:

$$f \oplus g = ((\text{dom } g) \triangleleft f) \cup g$$

There is notation in Z for injective, surjective and bijective functions. An injective function is one to one, i.e.

$$f(x) = f(y) \Rightarrow x = y.$$

A surjective function is onto, i.e.

$$\text{Given } y \in Y, \exists x \in X \text{ such that } f(x) = y.$$

A bijective function is one to one and onto, and it indicates that the sets X and Y can be put into one to one correspondence with one another. Z includes lambda calculus notation to define functions (λ -calculus is discussed in more detail in Chap. 12 of [ORg:16b]). For example, the function $\text{cube} = \lambda x:\mathbb{N} \cdot x * x * x$. Function composition $f; g$ is similar to relational composition.

8.5 Sequences

The type of all sequences of elements drawn from a set X is denoted by $\text{seq } X$. Sequences are written as $\langle x_1, x_2, \dots, x_n \rangle$, and the empty sequence is denoted by $\langle \rangle$. Sequences may be used to specify the changing state of a variable over time, with each element of the sequence representing the value of the variable at a discrete time instance.

Sequences are functions and a sequence of elements drawn from a set X is a finite function from the set of natural numbers to X . A partial finite function f from X to Y is denoted by $f: X \mapsto Y$. A finite sequence of elements of X is given by a finite function $f: \mathbb{N} \mapsto X$, and the domain of the function consists of all numbers between 1 and $\#f$ (where $\#f$ is the cardinality of f). It is defined formally as:

$$\text{seq } X == \{f: \mathbb{N} \mapsto X \mid \text{dom}f = 1.. \#f \bullet f\}$$

The sequence $\langle x_1, x_2, \dots, x_n \rangle$ above is given by:

$$\{1 \mapsto x_1, 2 \mapsto x_2, \dots, n \mapsto x_n\}.$$

There are various functions to manipulate sequences. These include the sequence concatenation operation. Suppose $\sigma = \langle x_1, x_2, \dots, x_n \rangle$ and $\tau = \langle y_1, y_2, \dots, y_m \rangle$ then:

$$\sigma \sqcap \tau = \langle x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m \rangle.$$

The head of a non-empty sequence gives the first element of the sequence.

$$\text{head } \sigma = \text{head} \langle x_1, x_2, \dots, x_n \rangle = x_1.$$

The tail of a non-empty sequence is the same sequence except that the first element of the sequence is removed.

$$\text{tail } \sigma = \text{tail} \langle x_1, x_2, \dots, x_n \rangle = \langle x_2, \dots, x_n \rangle.$$

Suppose $f: X \rightarrow Y$ and a sequence $\sigma: \text{seq } X$ then the function map applies f to each element of σ :

$$\text{map } f \sigma = \text{map } f \langle x_1, x_2, \dots, x_n \rangle = \langle f(x_1), f(x_2), \dots, f(x_n) \rangle.$$

The map function may also be expressed via function composition as:

$$\text{map } f \sigma = \sigma; f.$$

The reverse order of a sequence is given by the rev function:

$$\text{rev } \sigma = \text{rev} \langle x_1, x_2, \dots, x_n \rangle = \langle x_n, \dots, x_2, x_1 \rangle.$$

8.6 Bags

A bag is similar to a set except that there may be multiple occurrences of each element in the bag. A bag of elements of type X is defined as a partial function from the type of the elements of the bag to positive whole numbers. The definition of a bag of type X is:

$$\text{bag } X ::= X \rightarrow \mathbb{N}_1.$$

For example, a bag of marbles may contain 3 blue marbles, 2 red marbles and 1 green marble. This is denoted by $B = \llbracket b, b, b, g, r, r \rrbracket$. The bag of marbles is thus denoted by:

$$\text{bag } \textit{Marble} ::= \textit{Marble} \rightarrow \mathbb{N}_1.$$

The function count determines the number of occurrences of an element in a bag. For the example above, count *Marble* $b = 3$, and count *Marble* $y = 0$ since there are no yellow marbles in the bag. This is defined formally as:

$$\begin{aligned} \text{count bag } X \ y = 0 & \quad y \notin \text{bag } X \\ \text{count bag } X \ y = (\text{bag } X)(y) & \quad y \in \text{bag } X. \end{aligned}$$

An element y is in bag X if and only if y is in the domain of bag X .

$$y \text{ in bag } X \Leftrightarrow y \in \text{dom}(\text{bag } X).$$

The union of two bags of marbles $B_1 = \llbracket b, b, b, g, r, r \rrbracket$ and $B_2 = \llbracket b, g, r, y \rrbracket$ is given by $B_1 \uplus B_2 = \llbracket b, b, b, b, g, g, r, r, r, y \rrbracket$. It is defined formally as:

$$\begin{aligned} (B_1 \uplus B_2)(y) = B_2(y) & \quad y \notin \text{dom } B_1 \wedge y \in \text{dom } B_2 \\ (B_1 \uplus B_2)(y) = B_1(y) & \quad y \in \text{dom } B_1 \wedge y \notin \text{dom } B_2 \\ (B_1 \uplus B_2)(y) = B_1(y) + B_2(y) & \quad y \in \text{dom } B_1 \wedge y \in \text{dom } B_2. \end{aligned}$$

A bag may be used to record the number of occurrences of each product in a warehouse as part of an inventory system. It may model the number of items remaining for each product in a vending machine (Fig. 8.4).

$$\left| \begin{array}{l} \text{---}\Delta\textit{Vending Machine}\text{---} \\ \textit{stock} : \text{bag } \textit{Good} \\ \textit{price} : \textit{Good} \rightarrow \mathbb{N}_1 \\ \hline \text{dom } \textit{stock} \subseteq \text{dom } \textit{price} \\ \hline \end{array} \right.$$

Fig. 8.4 Specification of vending machine using bags

The operation of a vending machine would require other operations such as identifying the set of acceptable coins, checking that the customer has entered sufficient coins to cover the cost of the good, returning change to the customer and updating the quantity on hand of each good after a purchase. A more detailed examination is in [Dil:90].

8.7 Schemas and Schema Composition

The schemas in Z are visually striking and the specification is presented in two-dimensional graphic boxes. Schemas are used for specifying states and state transitions, and they employ notation to represent the before and after state (e.g. s and s' where s' represents the after state of s). The schemas group all relevant information that belongs to a state description.

There are a number of useful schema operations such as schema inclusion, schema composition and the use of propositional connectives to link schemas together. The Δ convention indicates that the operation affects the state, whereas the Ξ convention indicates that the state is not affected. These operations and conventions allow complex operations to be specified concisely and assist with the readability of the specification. Schema composition is analogous to relational composition and allows new schemas to be derived from existing schemas.

A schema name S_1 may be included in the declaration part of another schema S_2 . The effect of the inclusion is that the declarations in S_1 are now part of S_2 , and the predicates of S_1 and S_2 are joined together by conjunction. If the same variable is defined in both S_1 and S_2 , then it must be of the same type in both schemas.

$$\left| \begin{array}{l} \text{---}S_1\text{---} \\ x, y : \mathbb{N} \\ \hline x + y > 2 \\ \hline \end{array} \right| \quad \left| \begin{array}{l} \text{---}S_2\text{---} \\ S_1 ; z : \mathbb{N} \\ \hline z = x + y \\ \hline \end{array} \right|$$

The result is that S_2 includes the declarations and predicates of S_1 (Fig. 8.5):

Two schemas may be linked by propositional connectives such as $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \rightarrow S_2$ and $S_1 \leftrightarrow S_2$. The schema $S_1 \vee S_2$ is formed by merging the declaration parts of S_1 and S_2 and then combining their predicates by the logical \vee operator. For example, $S = S_1 \vee S_2$ yields (Fig. 8.6):

$$\left| \begin{array}{l} \text{---}S_2\text{---} \\ x, y : \mathbb{N} \\ z : \mathbb{N} \\ \hline x + y > 2 \\ z = x + y \\ \hline \end{array} \right|$$

Fig. 8.5 Schema inclusion

$$\frac{\begin{array}{l} \text{---S---} \\ x, y : \mathbb{N} \\ z : \mathbb{N} \\ \text{---} \\ x + y > 2 \vee z = x + y \\ \text{---} \end{array}}{\text{---}}$$

Fig. 8.6 Merging schemas ($S_1 \vee S_2$)

Schema inclusion and the linking of schemas use normalization to convert subtypes to maximal types, and predicates are employed to restrict the maximal type to the subtype. This involves replacing declarations of variables (e.g. $u: 1..35$ with $u: \mathbb{Z}$, and adding the predicate $u > 0$ and $u < 36$ to the predicate part of the schema).

The Δ and Ξ conventions are used extensively, and the notation $\Delta TempMap$ is used in the specification of schemas that involve a change of state. The notation $\Delta TempMap$ represents:

$$\Delta TempMap = TempMap \wedge TempMap'$$

The longer form of $\Delta TempMap$ is written as:

$$\frac{\begin{array}{l} \text{---}\Delta TempMap\text{---} \\ CityList, CityList' : \mathbb{P} \text{ City} \\ temp, temp' : City \leftrightarrow Z \\ \text{---} \\ \text{dom } temp = CityList \\ \text{dom } temp' = CityList' \\ \text{---} \end{array}}{\text{---}}$$

The notation $\Xi TempMap$ is used in the specification of operations that do not involve a change to the state.

$$\frac{\begin{array}{l} \text{---}\Xi TempMap\text{---} \\ \Delta TempMap \\ \text{---} \\ CityList = CityList' \\ temp = temp' \\ \text{---} \end{array}}{\text{---}}$$

Schema composition is analogous to relational composition, and it allows new specifications to be built from existing ones. It allows the after state variables of one schema to be related with the before variables of another schema. The composition of two schemas S and T ($S; T$) is described in detail in [Dil:90] and involves four steps (Table 8.1).

The example below should make schema composition clearer. Consider the composition of S and T where S and T are defined as follows:

Table 8.1 Schema composition

Step	Procedure
1.	Rename all <i>after</i> state variables in S to something new: S $ s^+ / s' $
2.	Rename all <i>before</i> state variables in T to the same new thing, i.e. T $ s^+ / s $
3.	Form the conjunction of the two new schemas: S $[s^+ / s'] \wedge T [s^+ / s]$
4.	Hide the variable introduced in step 1 and 2. S; T = (S $[s^+ / s'] \wedge T [s^+ / s]) \setminus (s^+)$

$$\left| \frac{-S_1 \wedge T_1 \text{---}}{x, x^+, x', y? : \mathbb{N}} \right| \quad \left| \frac{-S ; T \text{---}}{x, x', y? : \mathbb{N}} \right|$$

$$\left| \frac{x^+ = y? - 2}{x' = x^+ + 1} \right| \quad \left| \frac{\exists x^+ : \mathbb{N} \bullet (x^+ = y? - 2)}{x' = x^+ + 1} \right|$$

Fig. 8.7 Schema composition

$$\left| \frac{-S \text{---}}{x, x', y? : \mathbb{N}} \right| \quad \left| \frac{-T \text{---}}{x, x' : \mathbb{N}} \right|$$

$$\left| \frac{x' = y? - 2}{\text{---}} \right| \quad \left| \frac{x' = x + 1}{\text{---}} \right|$$

$$\left| \frac{-S_1 \text{---}}{x, x^+, y? : \mathbb{N}} \right| \quad \left| \frac{-T_1 \text{---}}{x^+, x' : \mathbb{N}} \right|$$

$$\left| \frac{x^+ = y? - 2}{\text{---}} \right| \quad \left| \frac{x' = x^+ + 1}{\text{---}} \right|$$

S_1 and T_1 represent the results of step 1 and step 2, with x' renamed to x^+ in S, and x renamed to x^+ in T. Step 3 and step 4 yield (Fig. 8.7):

Schema composition is useful as it allows new specifications to be created from existing ones.

8.8 Reification and Decomposition

A Z specification involves defining the state of the system and then specifying the required operations. The Z specification language employs many constructs that are not part of conventional programming languages, and a Z specification is therefore not directly executable on a computer. A programmer implements the formal specification, and mathematical proof may be employed to prove that a program meets its specification.

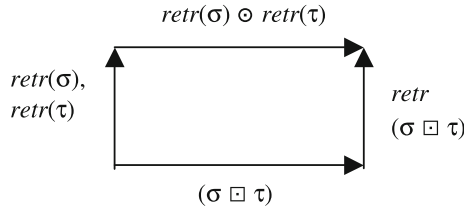


Fig. 8.8 Refinement commuting diagram

Often, there is a need to write an intermediate specification that is between the original Z specification and the eventual program code. This intermediate specification is more algorithmic and uses less abstract data types than the Z specification. The intermediate specification is termed the design and the design needs to be correct with respect to the specification, and the program needs to be correct with respect to the design. The design is a refinement (reification) of the state of the specification, and the operations of the specification have been decomposed into those of the design.

The representation of an abstract data type such as a set by a sequence is termed data reification, and data reification is concerned with the process of transforming an abstract data type into a concrete data type. The abstract and concrete data types are related by the retrieve function, and the retrieve function maps the concrete data type to the abstract data type. There are typically several possible concrete data types for a particular abstract data type (i.e. refinement is a relation), whereas there is one abstract data type for a concrete data type (i.e. retrieval is a function). For example, sets are often reified to unique sequences; and clearly more than one unique sequence can represent a set, whereas a unique sequence represents exactly one set.

The operations defined on the concrete data type are related to the operations defined on the abstract data type. That is, the commuting diagram property is required to hold (Fig. 8.8). That is, for an operation \sqsupset on the concrete data type to correctly model the operation \circ on the abstract data type the following diagram must commute, and the commuting diagram property requires proof. That is, it is required to prove that:

$$\text{retr}(\sigma \sqsupset \tau) = (\text{retr} \sigma) \circ (\text{retr} \tau)$$

In Z, the refinement and decomposition is done with schemas. It is required to prove that the concrete schema is a valid refinement of the abstract schema, and this gives rise to a number of proof obligations. It needs to be proved that the initial states correspond to one another, and that each operation in the concrete schema is correct with respect to the operation in the abstract schema, and also that it is applicable (i.e. whenever the abstract operation may be performed the concrete operation may also be performed).

8.9 Proof in Z

Mathematicians perform rigorous proof of theorems using technical and natural language. Logicians employ formal proofs to prove theorems using propositional and predicate calculus. Formal proofs generally involve a long chain of reasoning with every step of the proof justified. Rigorous proofs involve precise reasoning using a mixture of natural and mathematical language. Rigorous proofs [Dil:90] have been described as being analogous to high-level programming languages, whereas formal proofs are analogous to machine language.

A mathematical proof includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. Many proofs in formal methods such as Z are concerned with crosschecking on the details of the specification, or on the validity of the refinement step, or proofs that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and tool support is essential as proof by hand often contains errors or jumps in reasoning. Machine proofs are lengthy and largely unreadable; however, they provide extra confidence as every step in the proof is justified.

The proof of various properties about the programs increases confidence in its correctness.

8.10 Industrial Applications of Z

The Z specification language is one of the more popular formal methods, and it has been employed for the formal specification and verification of safety critical software. IBM piloted the Z formal specification language on the CICS (Customer Information Control System) project at its plant in Hursley, England.

Rolls Royce and Associates (RRA) developed a lifecycle suitable for the development of safety critical software, and the safety-critical lifecycle used Z for the formal specification and the CADiZ tool provided support for specification, and Ada was the target implementation language.

Logica employed Z for the formal verification of a smart card-based electronic cash system (the Mondex smart card) in the early 1990s. The smart card had an 8-bit microprocessor, and the objective was to formally specify both the high-level abstract security policy model and the lower-level concrete architectural design in Z, and to provide a formal proof of correspondence between the two.

Computer Management Group (CMG) employed Z for modelling data and operations as part of the formal specification of a movable barrier (the Maeslantkering), which is used to protect the port of Rotterdam from flooding. The decisions on opening and closing of the barrier are based on meteorological data provided by the computer system, and the focus of the application of formal methods was to the decision-making subsystem and its interfaces to the environment.

8.11 Review Questions

1. Describe the main features of the Z specification language.
2. Explain the difference between $\mathbb{P}_1 X$, $\mathbb{P} X$ and FX .
3. Give an example of a set derived from another set using set comprehension. Explain the three main parts of set comprehension in Z .
4. Discuss the applications of Z and which areas have benefited most from their use? What problems have arisen?
5. Give examples to illustrate the use of domain and range restriction operators and domain and range anti-restriction operators with relations in Z .
6. Give examples to illustrate relational composition.
7. Explain the difference between a partial and total function, and give examples to illustrate function override.
8. Give examples to illustrate the various operations on sequences including concatenation, head, tail, map and reverse operations.
9. Give examples to illustrate the various operations on bags.
10. Discuss the nature of proof in Z and tools to support proof.
11. Explain the process of refining an abstract schema to a more concrete representation, the proof obligations that are generated, and the commuting diagram property.

8.12 Summary

Z is a formal specification language that was developed in the early 1980s at Oxford University in England. It has been employed in both industry and academia, and it was used successfully on the IBM's CICS project. Its specifications are mathematical, and this leads to more rigorous software development. Its mathematical approach allows properties to be proved about the specification, and any gaps or inconsistencies in the specification may be identified.

Z is a model-oriented approach and an explicit model of the state of an abstract machine is given, and the operations are defined in terms of their effect on the state. Its main features include a mathematical notation that is similar to VDM and the schema calculus. The latter consists essentially of boxes and is used to describe operations and states.

The schema calculus enables schemas to be used as building blocks to form larger specifications. It is a powerful means of decomposing a specification into smaller pieces, and helps with the readability of Z specifications, as each schema is small in size and self-contained.

Z is a highly expressive specification language, and it includes notation for sets, functions, relations, bags, sequences, predicate calculus and schema calculus. Z specifications are not directly executable as many of its data types and constructs are not part of modern programming languages. Therefore, there is a need to refine the Z specification into a more concrete representation and prove that the refinement is valid.

9.1 Introduction

VDM dates from work done by the IBM research laboratory in Vienna in the late 1960s. Their aim was to specify the semantics of the PL/1 programming language. This was achieved by employing the Vienna Definition Language (VDL), taking an operational semantic approach; that is, the semantics of a language are determined in terms of a hypothetical machine which interprets the programs of that language [BjJ:82]. Later work led to the Vienna Development Method (VDM) with its specification language, Meta IV.¹ This concerned itself with the denotational semantics of programming languages; that is, a mathematical object (set, function, etc.) is associated with each phrase of the language [BjJ:82]. The mathematical object is the *denotation* of the phrase. The initial application of VDM was to program language semantics, whereas today, VDM is mainly employed to formally specify software, and it includes a rigorous method for software specification and development.

The IBM Vienna group was broken up in the mid-1970s, and this led to a diaspora of the project team, and it led to the formation of different schools of the VDM in multiple locations. These include the “Danish school” led by Dines Bjorner; the English school led by Cliff Jones; and the Polish school led by Andrez Blikle as described in [Mac:90]. Further work on VDM and Meta IV continued in the 1980s, and standards for VDM (VDM-SL) appeared in the 1990s.

VDM is a “*model-oriented approach*”, and this means that an explicit model of the state of an abstract machine is given, and operations are defined in terms of this state. Operations may act on the system state, taking inputs and producing outputs and a new system state. Operations are defined in a precondition and postcondition style. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant.

¹Meta IV was a pun on metaphor.

VDM uses keywords to distinguish different parts of the specification; for example, preconditions and postconditions are introduced by the keywords *pre* and *post*, respectively. In keeping with the philosophy that formal methods specifies *what* a system does as distinct from *how*, VDM employs postconditions to stipulate the effect of the operation on the state. The previous state is then distinguished by employing *hooked variables*; for example, v^- and the postcondition specify the new state (*defined by a logical predicate relating the prestate to the poststate*) from the previous state.

VDM is more than its specification language Meta IV (called VDM-SL in the standardization of VDM), and it is, in fact, a development method, with rules to verify the steps of development. The rules enable the executable specification, i.e. the detailed code, to be obtained from the initial specification via refinement steps. In another words, we have a sequence $S = S_0, S_1, \dots, S_n = E$ of specifications, where S is the initial specification, and E is the final (executable) specification.

$$S = S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \dots \sqsubseteq S_n = E$$

Retrieval functions enable a return from a more concrete specification, to the more abstract specification. The initial specification consists of an initial state, a system state and a set of operations. The system state is a particular domain, where a domain is built out of primitive domains such as the set of natural numbers, or constructed from primitive domains using domain constructors such as Cartesian product, disjoint union. A domain-invariant predicate may further constrain the domain, and a *type* in VDM reflects a domain obtained in this way. Thus, a type in VDM is more specific than the signature of the type and thus represents values in the domain defined by the signature, which satisfy the domain invariant. In view of this approach to types, it is clear that VDM types may not be “statically typed checked”.

VDM specifications are structured into modules, with a module containing the module name, parameters, types, operations etc. Partial functions arise naturally in computer science. The problem is that many functions, especially recursively defined functions can be undefined, or fail to terminate for some arguments in their domain. VDM addresses partial functions by employing non-standard logical operators, namely the logic of partial functions (LPFs) which can deal with undefined operands. This was developed by Cliff Jones and is discussed later in the chapter.

The Boolean expression $T \vee \perp = \perp \vee T = \text{true}$; that is, the truth-value of a logical *or* operation is true if at least one of the logical operands is true, and the undefined term is treated as a do not care value. The similarities and differences between Z and VDM (the two most widely used formal methods) are summarized in Table 9.1.

Example 9.1 The following is a very simple example of a VDM specification and is adapted from [InA:91]. It is a simple library system that allows borrowing and returning of books. The data types for the library system are first defined, and the

Table 9.1 Similarities and differences between VDM and Z

VDM is a development method including a specification language, whereas Z is a specification language only
Constraints may be placed on types in VDM specifications but not in Z specifications
Z is structured into schemas and VDM into modules
The schema calculus is part of Z
Relations are part of Z but not of VDM
VPreconditions are not separated out in Z specifications
DM employs the logic of partial functions (3-valued logic), whereas Z is a classical 2-valued logic

operation to borrow a book is then defined. It is assumed that the state is made up of three sets, and these are the set of books on the shelf, the set of books which are borrowed, and the set of missing books. These sets are mutually disjoint.

The effect of the operation to borrow a book is to remove the book from the set of books on the shelf and to add it to the set of borrowed books. The reader is referred to [InA:91] for a detailed explanation.

types

$Bks = Bkd-id$ set

state *Library* of

On-shelf : Bks

Missing : Bks

Borrowed : Bks

inv *mk-Library* (*os*, *mb*, *bb*) $\underline{\Delta}$ *is-disj*($\{os, mb, bb\}$)

end

borrow (*b*: $Bkd-id$)

ex wr *on-shelf*, *borrowed* : Bks

pre $b \in on-shelf$

post $on-shelf = on-shelf^- - \{b\} \wedge$

$borrowed = borrowed^- \cup \{b\}$

A VDM specification consists of

- Type definitions
- State Definitions
- Invariant for the system
- Definition of the operations of the system

Table 9.2 Built-in types in VDM

Set	Name	Elements
\mathbf{B}	Boolean	{true, false}
\mathbf{N}	Naturals	{0, 1, ...}
\mathbf{N}_1	Naturals (excluding 0)	{1, 2, ...}
\mathbf{Z}	Integers	{...-1, 0, 1, ...}
\mathbf{Q}	Rational numbers	$\{^p/q : p, q \in \mathbb{Z} \ q \neq 0\}$
\mathbf{R}	Real numbers	

The notation *Bkd-id* set specifies that *Bks* is a set of *Bkd-ids*, e.g. $Bks = \{b_1, b_2, \dots, b_n\}$. The invariant specifies the property that must remain true for the library system. The *borrow* operation is defined using preconditions and postconditions. The notation “ext wr” indicates that the *borrow* operation affects the state, whereas the notation “ext rd” indicates an operation that does not affect the state.

VDM is a widely used formal method and has been used in industrial strength projects as well as by the academic community. These include security-critical systems and safety critical sectors such as the railway industry. There is tool support available, for example, the IFAD VDM-SL toolbox² and the open-source Overture IDE tool. There are several variants of VDM, including VDM++, an object-oriented extension of VDM and the Irish school of the VDM, which is discussed in the next chapter.

9.2 Sets

Sets are a key building block of VDM specifications. A set is a collection of objects that contain no duplicates. The set of all even natural numbers less than or equal to 10 is given by:

$$S = \{2, 4, 6, 8, 10\}$$

There are a number of in-built sets that are part of VDM including (Table 9.2).

The empty set is a set with no members and is denoted by $\{ \}$. The membership of a set S is denoted by $x \in S$. A set S is a subset of a set T if whenever $x \in S$, then $x \in T$. This is written as $S \subseteq T$. The union of two sets S and T is given by $S \cup T$. The intersection of two sets S and T is given by $S \cap T$.

Sets may be specified by enumeration (as in $S = \{2, 4, 6, 8, 10\}$). However, set enumeration is impractical for large sets. The more general form of specification of sets is termed set comprehension and is of the form:

²The IFAD Toolbox has been renamed to VDMTools (as IFAD sold the VDM Tools to CSK in Japan).

{set membership | predicate}

For example, the specification of the set $T = \{x \in \{2, 4, 6, 8, 10\} \mid x > 5\}$ denotes the set $T = \{6, 8, 10\}$. The set $Q = \{x \in \mathbb{N} \mid x > 5 \wedge x < 8\}$ denotes the set $Q = \{6, 7\}$.

The set of all finite subsets of a set $S = \{1, 2\}$ is given by:

$$\mathcal{F}S = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$$

The notation $S : \text{set } A$ denotes that S is a set, with each element in S drawn from A . For example, for $A = \{1, 2\}$, the valid values of S are $S = \{\}, S = \{1\}, S = \{2\}$, or $S = \{1, 2\}$.

The set difference of two sets S and T is given by $S - T$ where:

$$S - T = \{x \in S \mid x \in S \wedge x \notin T\}$$

Given $S = \{2, 4, 6, 8, 10\}$ and $T = \{4, 8, 12\}$ then $S - T = \{2, 6, 10\}$.

Finally, the distributed union and intersection operators are considered. These operators are applied to a set of sets.

$$\cap \{S_1, S_2, \dots, S_n\} = S_1 \cap S_2 \cap \dots \cap S_n$$

$$\cup \{S_1, S_2, \dots, S_n\} = S_1 \cup S_2 \cup \dots \cup S_n$$

The cardinality of a set S is given by $\text{card } S$. This gives the number of elements in the set; for example, $\text{card } \{1, 3\} = 2$. The notation $Bks = Bkd\text{-id-set}$ in Example 9.1 above specifies that Bks is a set of $Bkd\text{-ids}$, i.e. $Bks = \{b_1, b_2, \dots, b_n\}$.

9.3 Sequences

Sequences are used frequently in VDM specifications (e.g. in the modelling of stacks). A sequence is a collection of items that are ordered in a particular way. Duplicate items are allowed for sequences, whereas duplicate elements are not meaningful for sets (unless we are dealing with multi-sets or bags). A set may be refined to a sequence of unique elements.

A sequence of elements x_1, x_2, \dots, x_n is denoted by $[x_1, x_2, \dots, x_n]$, and the empty sequence is denoted by $[\]$. Given a set S , then S^* denotes the set of all finite sequences constructed from the elements of S .

The length of a sequence is given by the len operator and

$$\begin{aligned} \text{len } [\] &= 0 \\ \text{len } [1, 2, 6] &= 3 \end{aligned}$$

The `hd` operation gives the first element of the sequence. It is applied to non-empty sequences only:

$$\begin{aligned}\text{hd } [x] &= x \\ \text{hd } [x, y, z] &= x\end{aligned}$$

The `tl` operation gives the remainder of a sequence after the first element of the sequence has been removed. It is applied to non-empty sequences only:

$$\begin{aligned}\text{tl } [x] &= [] \\ \text{tl } [x, y, z] &= [y, z]\end{aligned}$$

The `elems` operation gives the elements of a sequence. It is applied to both empty and non-empty sequences:

$$\begin{aligned}\text{elems } [] &= \{ \} \\ \text{elems } [x, y, z] &= \{x, y, z\}\end{aligned}$$

The `inds` operation is applied to both empty and non-empty sequences. It returns the set $\{1, 2, \dots, n\}$ where n is the number of elements in the sequence.

$$\begin{aligned}\text{inds } [] &= \{ \} \\ \text{inds } [x, y, z] &= \{1, 2, 3\} \\ \text{inds } s &= \{1, .. \text{len } s\}\end{aligned}$$

Two sequences may be joined together by the concatenation operator:

$$\begin{aligned}[] \curvearrowright [] &= [] \\ [x, y, z] \curvearrowright [a, b] &= [x, y, z, a, b] \\ [x, y] \curvearrowright [] &= [x, y]\end{aligned}$$

Two sequences s_1 and s_2 are equal if:

$$s_1 = s_2 \Leftrightarrow (\text{len } s_1 = \text{len } s_2) \wedge (\forall i \in \text{inds } s_1)(s_1(i) = s_2(i))$$

Sequences may be employed to specify a stack. For example, a stack of (up to 100) integers is specified as (Table 9.3):

Table 9.3 Specification of a stack of integers

state Z-stack of
stk : \mathbb{Z}^*
inv-Z-stack : $\mathbb{Z}^* \rightarrow \mathbf{B}$
inv-Z-stack (stk) $\underline{\Delta}$ len stk ≤ 100
init-mk-Z-stack (stk) $\underline{\Delta}$
stk = []
end

The push operation is then specified in terms of preconditions/postconditions as follows.

$$\begin{aligned} & \text{push}(z : \mathbb{Z}) \\ & \text{pre len stk} < 100 \\ & \text{post stk} = [z] \curvearrowright \text{stk}^{\leftarrow} \end{aligned}$$

9.4 Maps

Maps (also called partial functions) are frequently employed for modelling and specifications in VDM. A map is used to relate the members of two sets X and Y such that an element from the first set X is associated with (at most) one element in the second set Y . The map from X to Y is denoted by:

$$f : T = X \rightarrow^m Y$$

The domain of the map f is a subset of X and the range is a subset of Y . An example of a map declaration is:

$$f : \{Names \rightarrow^m AccountNmr\}$$

The map f may take on the values:

$$\begin{aligned} f &= \{ \} \\ f &= \{eithne \mapsto 231, fred \mapsto 315\} \end{aligned}$$

The domain and range of f are given by:

$$\begin{aligned} \text{dom } f &= \{eithne, fred\} \\ \text{rng } f &= \{231, 315\} \end{aligned}$$

The map overwrites operator $f \dagger g$ gives a map that contains all the maplets in the second operand together with the maplets in the first operand that are not in the domain of the second operand.³

$$\begin{aligned} \text{For } g &= \{eithne \mapsto 412, aisling \mapsto 294\} \text{ then} \\ f \dagger g &= \{eithne \mapsto 412, aisling \mapsto 294, fred \mapsto 315\} \end{aligned}$$

The map restriction operator has two operands: the first operand is a set, whereas the second operand is a map. It forms the map by extracting those maplets that have the first element equal to a member of the set. For example,

³ $f \dagger g$ is the VDM notation for function override. The notation $f \oplus g$ is employed in Z.

$$\{eithne\} \triangleleft \{\langle eithne \mapsto 412, aisling \mapsto 294, fred \mapsto 315 \rangle\} = \{eithne \mapsto 412\}$$

The map deletion operator has two operands: the first operand is a set, whereas the second operand is a map. It forms the map by deleting those maplets that have the first element equal to a member of the set. For example,

$$\{eithne, fred\} \triangleleft \{\langle eithne \mapsto 412, aisling \mapsto 294, fred \mapsto 315 \rangle\} = \{aisling \mapsto 294\}$$

Total maps are termed functions, and a total function f from a set X to a set Y is denoted by:

$$f : X \rightarrow Y$$

A partial function (map) $f : X \rightarrow^m Y$ arises frequently in specifications and may be undefined for some values in X .

9.5 Logic of Partial Functions in VDM

We discussed the logic of partial functions (LPFs) in Chap. 7, which is used to deal with terms that may be undefined. LPF is a three-valued logic that was developed by Cliff Jones [Jon:86], and a logical term may be true, false or undefined.

The conjunction of P and Q ($P \wedge Q$) is true when both P and Q are true; false if one of P or Q is false and undefined otherwise. The disjunction of P and Q ($P \vee Q$) is true if one of P or Q is true; false if both P and Q are false and undefined otherwise. The implication operation ($P \rightarrow Q$) is true when P is false or when Q is true; it is false when P is true and Q is false and undefined otherwise.⁴

The equivalence operation ($P \leftrightarrow Q$) is true when both P and Q are true or false; it is false when P is true and Q is false (or vice versa); it is undefined otherwise. The not operator (\neg) is a unary operator such $\neg A$ is true when A is false, false when A is true and undefined when A is undefined.

It is clear from the truth table definitions in Chap. 7 that the result of the operation may be known immediately after knowing the value of one of the operands (e.g. disjunction is true if P is true irrespective of the value of Q). The law of the excluded middle, i.e. $A \vee \neg A = \text{true}$, does not hold in the 3-valued logic of partial functions.

⁴The problem with 3-valued logic is that they are less intuitive than classical 2-valued logic.

9.6 Data Types and Data Invariants

Larger specifications often require more complex data types. The VDM specification language allows composite data types to be created from their underlying component data types. For example, [InA:91] the composite data type *Date* is defined as follows (Table 9.4).

A make function is employed to construct a date from the components of the date; that is, the *mk-Date* function takes three numbers as arguments and constructs a date from them.

$$mk-Date : \{2000, \dots, 3000\} \times \{1, \dots, 12\} \times \{1, \dots, 31\} \rightarrow Date$$

For example, the date of 5 August 2004 is constructed as follows:

$$mk-Date(2004, 8, 5)$$

Selectors are employed to take a complex data type apart into its components. The selectors employed for date are day, month and year. Hence, the selection of the year component in the date of 5 August 2004 is:

$$mk-Date : (2004, 8, 5).year = 2004$$

The reader will note that the definition of the *Date* data type above allows invalid dates to be present, e.g. 29 February 2001 and 31 November 2004. Hence, what is required is a predicate to restrict elements of the data type to be valid dates. This is achieved by a data invariant (Table 9.5).

Any operation that affects the date will need to preserve the data invariant. This gives rise to a proof obligation for each operation that affects the date.

Table 9.4 Composite data types in VDM

<i>Date</i> = compose <i>Date</i> of
<i>year</i> : {2000, ..., 3000}
<i>month</i> : {1, ..., 12}
<i>day</i> : {1, ..., 31}
end

Table 9.5 Composite data invariant for composite date datatype

<i>Inv-Date</i> : <i>Date</i> → B
<i>Inv-Date</i> (<i>dt</i>) Δ
let <i>mk-Date</i> (<i>yr</i> , <i>md</i> , <i>dy</i>) = <i>dt</i> in
(<i>md</i> ∈ {1, 3, 5, 7, 8, 10, 12} ∧ <i>dy</i> ∈ {1, ..., 31})
∨ (<i>md</i> ∈ {4, 6, 9, 11} ∧ <i>dy</i> ∈ {1, ..., 30})
∨ (<i>md</i> = 2 ∧ <i>isleapyear</i> (<i>yr</i>) ∧ <i>dy</i> ∈ {1, ..., 29})
∨ (<i>md</i> = 2 ∧ ¬ <i>isleapyear</i> (<i>yr</i>) ∧ <i>dy</i> ∈ {1, ..., 28})

9.7 Specification in VDM

An abstract machine (sometimes called object) consists of the specification of a data type together with the operations on the data type. The production of a large specification involves [InA:91]:

1. Identifying and specifying the abstract machines.
2. Defining how these machines fit together and are controlled to provide the required functionality.

The abstract machines may be identified using design tools such as data flow diagrams and object-oriented design. Once the abstract machines have been identified, there are then two further problems to be addressed.

1. How are the abstract machines to be used (e.g. users or other programs).
2. How are the abstract machines to be implemented in code.

VDM-SL specifications are like programs except that they are not executable. However, one important difference is that there are no side effects in VDM-SL expressions as in imperative programs. The VDM specification is structured into type definitions, state definitions, an invariant for the system, the initial state and the definition of the operations of the system (Table 9.6).

The whole of the development process is based on the formal specification, and it is therefore essential that the specification is correct. A description of the development of the specification of the library system is presented in [InA:91].

Table 9.6 Structure of VDM specification

Name	Description
Type definitions	The type definitions specify the data types employed. These include the built-in sets, or sets constructed from existing sets. A domain-invariant predicate may further constrain the definition. A type in VDM is more specific than the signature of the type and represents values in the domain defined by the signature, which satisfy the domain invariant
State definitions	This is the definition of the collection of stored data. The operations access/modify the data
(Data-) invariant for the system	This describes a condition that must be true for the state throughout the execution of the system
Initial value of the state	This specifies the initial value of the state
Definition of operations	The operations on the state are defined. These are defined in terms of preconditions and postconditions. The keywords “rd” and “wr” indicate whether the operation changes the state

9.8 Refinement in VDM

The development of executable code from a VDM specification involves breaking down the specification into smaller specifications (each smaller specification defines an easier problem) [InA:91]. Each smaller specification is then tackled (this may involve even smaller subspecifications) until eventually the implementation of the code that satisfies each smaller specification is trivial (as well as the corresponding proofs of correctness). The code fragments are then glued together using the programming language constructs of the semicolon, the conditional statement and the while loop.

At each step of the process, a proof of correctness is conducted to ensure that the refinement is valid. The approach allows a large specification to be broken down to a smaller set of specifications that can be translated into code. It involves deriving equivalent specifications to existing specifications, where a specification OP' is equivalent to a specification OP if any program that satisfies OP' also satisfies OP . The formal definition of equivalence is:

1. $\forall i \in \text{State} . \text{pre-Op}(i) \Rightarrow \text{pre-OP}'(i)$
2. $\forall i, o \in \text{State} . \text{pre-Op}(i) \wedge \text{post-Op}'(i, o) \Rightarrow \text{post-OP}(i, o)$

The idea of a program satisfying its specification can be expanded to a specification satisfying a specification as follows:

OP' sat OP if

1. $\forall i \in \text{State} . \text{pre-Op}(i) \Rightarrow \text{pre-OP}'(i)$
2. $\forall i, o \in \text{State} . \text{pre-Op}(i) \wedge \text{post-Op}'(i, o) \Rightarrow \text{post-OP}(i, o)$
3. $\forall i \in \text{State} . \text{pre-Op}'(i) \Rightarrow \exists o \in \text{State} . \text{post-OP}'(i, o)$

The formal definition requires that whenever an input satisfies the precondition of OP , then it must also satisfy the precondition of OP' . Further, the two specifications must agree on an answer for any input state variables that satisfy the precondition for OP . Finally, the third part expresses the idea of a specification terminating (similar to a program terminating). It expresses the requirement that the specification is implementable.

The production of a working program that satisfies the specification is evidence that a specification is satisfiable. There is a danger that the miracle program could be introduced while carrying out a program refinement. The miracle program is a program that has no implementable specification:

miracle
pre true
post false

Clearly, an executable version of miracle is not possible as the miracle program must be willing to accept any input and produce no output. Refinement is a weaker form of satisfaction (and allows the *miracle* program). It is denoted by the \sqsubseteq operator.

$$A \text{ sat } B \Rightarrow B \sqsubseteq A$$

$$A \sqsubseteq B \text{ and } B \text{ is implementable} \Rightarrow B \text{ sat } A$$

$$S \sqsubseteq R_1 \sqsubseteq R_2 \sqsubseteq \dots \sqsubseteq R_n \sqsubseteq p \wedge p \text{ is executable} \Rightarrow p \text{ sat } S$$

9.9 Industrial Applications of VDM

VDM is one of the oldest formal methods, and it was developed by IBM at its research laboratory in Vienna. The VDM specification language was initially used to specify the semantics of the PL/I programming language, and it was later applied to the formal specification and verification of software systems.

VDM++ (the object-oriented version of VDM) has been applied to the formal specification of a real-time information system for tracking and tracing rail traffic (the CombiCom project provided a real-time information system for the Rotterdam, Cologne, Verona rail freight corridor). VDM++ was used for the formal specification and Ada for the implementation.

VDM++ was employed for the formal specification of a new generation of IC chip developed by FeliCa Networks in Japan. A large number VDM++ test cases were generated and executed using the VDM Tools Interpreter (formerly the IFAD Tools). The VDM Tools also provided test coverage information (82% of VDM++ model covered) after the execution of the test cases, and the remaining parts of the model were manually inspected.

VDM-SL has been employed in domains that are unrelated to computer science, and one interesting application is its application to the formal specification of the single transferable vote (STV) algorithm for the Church of England [HB:96].

9.10 Review Questions

1. Describe the main features of VDM.
2. Explain the difference between a partial and a total function in VDM and give examples of each.
3. Explain the difference between a set and a sequence in VDM.
4. Discuss the applications of VDM to industry.

5. Explain the map restriction and deletion operators in VDM and give examples of them.
6. Explain how an invariant may be used in VDM to restrict the values in a data type.
7. Describe the process of specification and development with VDM.
8. Give examples to illustrate the various operations on sequences in VDM.
9. Discuss the nature of proof in VDM and the tools available to support proof.
10. Explain the process of refinement VDM, the proof obligations that are generated and the commuting diagram property.

9.11 Summary

VDM dates from work done by the IBM research laboratory in Vienna in the late 1960s. It includes a formal specification language (originally called Meta IV) and a method to develop high-quality software. The Vienna group was broken up in the mid-1970s, and this led to the formation of different schools of the VDM in various locations. Further work on VDM and Meta IV continued in the 1980s and standards for VDM (VDM-SL) appeared in the 1990s.

VDM is a “*model-oriented approach*”, and this means that an explicit model of the state of an abstract machine is given, and operations are defined in terms of this state. Operations are defined in a precondition and postcondition style. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. VDM employs postconditions to stipulate the effect of the operation on the state. The postcondition specifies the new state using a predicate that relates the prestate to the poststate.

VDM is both a specification language and a development method. Its method provides rules to verify the steps of development and enable the executable specification, i.e. the detailed code, to be obtained from the initial specification via refinement steps.

$$S = S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \cdots \sqsubseteq S_n = E$$

Retrieval functions enable a return from a more concrete specification, to the more abstract specification. The initial specification consists of an initial state, a system state and a set of operations.

VDM specifications are structured into modules, with a module containing the module name, parameters, types and operations. VDM employs the logic of partial functions (LPFs) to deal with undefined operands.

VDM has been used in industrial strength projects as well as by the academic community. There is tool support available, for example, the IFAD VDM-SL toolbox and the open-source Overture IDE tool. There are several variants of VDM, including VDM++, an object-oriented extension of VDM and the Irish school of the VDM, which is discussed in the next chapter

10.1 Introduction

The Irish School of VDM is a variant of standard VDM and is characterized by [Mac:90] its constructive approach, classical mathematical style and its terse notation. In particular, this method combines the *what* and *how* of formal methods in that its terse specification style stipulates in concise form *what* the system should do, and furthermore, the fact that its specifications are constructive (or functional) means that the *how* is included with the *what*. However, it is important to qualify this by stating that the *how* as presented by VDM^{*} is not directly executable, as several of its mathematical data types have no corresponding structure in high-level programming languages or functional programming languages. Thus, a conversion or reification of the specification into a functional or higher-level language must take place to ensure a successful execution. It should be noted that the fact that a specification is constructive is no guarantee that it is a good implementation strategy, if the construction itself is naive. This issue is considered in [Mac:90] (pp. 135–7), and the example considered is the construction of the Fibonacci series.

The Irish school follows a similar development methodology as in standard VDM and is a model-oriented approach. The initial specification is presented, with initial state and operations defined. The operations are presented with preconditions; however, no postcondition is necessary as the operation is “functionally”, i.e., explicitly constructed. Each operation has an associated proof obligation: if the precondition for the operation is true and the operation is performed, then the system invariant remains true after the operation. The proof of invariant preservation normally takes the form of *constructive proofs*. This is especially the case for *existence proofs* in that the philosophy of the school is to go further than to provide a theoretical proof of existence, rather the aim is to demonstrate existence constructively.

The emphasis is on constructive existence and the school avoids the existential quantifier of predicate calculus. In fact, reliance on logic in proof is kept to a minimum, and emphasis instead is placed on equational reasoning rather than on applying the rules of predicate calculus. Special emphasis is placed on studying algebraic structures and their morphisms, and structures with nice algebraic properties are sought. One such structure is the monoid, which has closure, associativity and a unit element. The monoid is a very common structure in computer science, and thus, it is appropriate to study and understand it. The concept of isomorphism is powerful, reflecting that two structures are essentially identical, and thus, we may choose to work with either, depending on which is more convenient for the task in hand.

The school has been influenced by the work of Polya and Lakatos. The former [Pol:57] advocated a style of problem solving characterized by solving a complex problem by first considering an easier subproblem, and considering several examples, which generally leads to a clearer insight into solving the main problem. Lakatos's approach to mathematical discovery [Lak:76] is characterized by heuristic methods. A primitive conjecture is proposed and if global counter-examples to the statement of the conjecture are discovered, then the corresponding "hidden lemma" for which this global counter-example is a local counter-example is identified and added to the statement of the primitive conjecture. The process repeats, until no more global counter-examples are found. A sceptical view of absolute truth or certainty is inherent in this.

Partial functions are the norm in VDM^* , and as in standard VDM, the problem is that recursively defined functions may be undefined, or fail to terminate for several of the arguments in their domain. The logic of partial functions (LPFs) is avoided, and instead, care is taken with recursive definitions to ensure termination is achieved for each argument. This is achieved by ensuring that the recursive argument is strictly decreasing in each recursive invocation. The \perp symbol is typically used in the Irish school to represent *undefined or unavailable* or *do not care*. Academic and industrial projects have been conducted using the method of the Irish school, but tool support is limited.

Example The following is the equivalent VDM^* specification of the earlier example of a simple library presented in standard VDM.

$$Bks = \mathbb{P} Bkd-id$$

$$Library = (Bks \times Bks \times Bks)$$

$$Os, Ms, Bw \in Bks$$

$$inv-Library (Os, Ms, Bw) \underline{\Delta} \begin{aligned} Os \cap Ms &= \emptyset \\ \wedge Os \cap Bw &= \emptyset \\ \wedge Bw \cap Ms &= \emptyset \end{aligned}$$

$$Bor : Bkd-id \rightarrow (Bks \times Bks) \rightarrow (Bks \times Bks)$$

$$Bor \llbracket [b] \rrbracket (Os, Bw) \underline{\Delta} (\llbracket [b] \rrbracket Os, Bw \cup \{b\})$$

$$\begin{aligned} \text{pre-Bor} &: Bkd-id \rightarrow (Bks \times Bks) \rightarrow \mathbf{B} \\ \text{pre-Bor} \llbracket [b] \rrbracket (Os, Bw) &\triangleq \chi \llbracket [b] \rrbracket Os \end{aligned}$$

There is, of course, a proof obligation to prove that the *Bor* operation preserves the invariant; that is, the three sets of borrowed, missing or on the shelf remain disjoint after the execution of the operation. Proof obligations require a mathematical proof by hand or a machine-assisted proof to verify that the invariant remains satisfied after the operation.

$$\begin{aligned} \text{pre-Bor} \llbracket [b] \rrbracket (Os, Bw) \wedge ((Os', Bw') = \text{Bor} \llbracket [b] \rrbracket (Os, Bw)) \\ \Rightarrow \text{inv-Library} (Os', Ms', Bw') \end{aligned}$$

We will discuss the notation used in VDM^* in later sections.

10.2 Mathematical Structures and Their Morphisms

The Irish school of VDM uses mathematical structures for the modelling of systems and to conduct proofs. There is an emphasis on identifying useful structures that will assist modelling and constructing new structures from existing ones. Some well-known structures used in VDM^* include:

- Semi-groups
- Monoids

A semi-group is a structure A with a binary operation $*$ such that the closure and associativity properties hold:

$$\begin{aligned} a * b \in A & \quad \forall a, b \in A \\ (a * b) * c = a * (b * c) & \quad \forall a, b, c \in A \end{aligned}$$

Examples of semi-groups include the natural numbers under addition, non-empty sets under the set union operation and non-empty sequences under concatenation. A semi-group is commutative if:

$$a * b = b * a \quad \forall a, b \in A.$$

A monoid M is a semi-group that has the additional property that there is an identity element $u \in M$ such that:

$$\begin{array}{ll}
a * b \in M & \forall a, b \in M \\
(a * b) * c = a * (b * c) & \forall a, b, c \in M \\
a * u = a = u * a & \forall a \in M
\end{array}$$

Examples of monoids include the Integers under addition, sets under the set union operation and non-empty sequences under concatenation. The identity element is 0 for the integers, the empty set \emptyset for set union and the empty sequence Λ for sequence concatenation. A monoid is commutative if $a * b = b * a \forall a, b \in M$. A monoid is denoted by $(M, *, u)$.

A function $h : (M, \oplus, u) \rightarrow (N, \otimes, v)$ is structure preserving (morphism) between two monoids (M, \oplus, u) and (N, \otimes, v) if the same result is obtained by either:

1. Evaluating the expression in M and then applying h to the result.
2. Applying h to each element of M and evaluating the result under \otimes .

A monoid homomorphism $h : (M, \oplus, u) \rightarrow (N, \otimes, v)$ is expressed in the commuting diagram below. It requires that the commuting diagram property holds and that the image of the identity of M is the identity of N (Fig. 10.1).

$$\begin{array}{l}
h(m_1 \oplus m_2) = h(m_1) \otimes h(m_2) \quad \forall m_1, m_2 \in M \\
h(u) = v
\end{array}$$

A morphism from $h : (M, \oplus, u) \rightarrow (M, \oplus, u)$ is termed an endomorphism.

Examples Consider the monoid of sequences $(\Sigma^*, \cap, \Lambda)$ ¹ and the monoid of natural numbers $(\mathbb{N}, +, 0)$. Then, the function *len* that gives the length of a sequence is a monoid homomorphism from $(\Sigma^*, \cap, \Lambda)$ to $(\mathbb{N}, +, 0)$. Clearly, $len(\Lambda) = 0$ and the commuting diagram property hold (Fig. 10.2):

The second example considered is from the monoid of sequences to the monoid of sets under set union. Then, the function *elems* gives the elements of a sequence is a monoid homomorphism from $(\Sigma^*, \cap, \Lambda)$ to $(\mathbb{P}\Sigma, \cup, \emptyset)$. Clearly, $elems(\Lambda) = \emptyset$ and the commuting diagram property holds.

Consider the set removal operation $\triangleleft[[S]]$ on the monoid of sets under set union. Then, the removal operation is a monoid endomorphism from $(\mathbb{P}\Sigma, \cup, \emptyset)$ to $(\mathbb{P}\Sigma, \cup, \emptyset)$ (Fig. 10.3).

$$\begin{array}{l}
\triangleleft[[S]](S_1 \cup S_2) = \triangleleft[[S]](S_1) \cup \triangleleft[[S]](S_2) \\
\triangleleft[[S]](\emptyset) = \emptyset
\end{array}$$

Set restriction $(\triangleleft[[S]])$ is also an endomorphism on $(\mathbb{P}\Sigma, \cup, \emptyset)$.

¹One striking feature of the Irish VDM notation is its use of the Greek alphabet, and Σ^* defines the monoid of sequences over the alphabet Σ . The concatenation operator is denoted by \cap and the empty sequence is denoted by Λ .

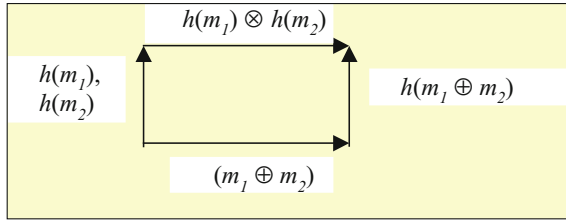


Fig. 10.1 Monoid homomorphism

Fig. 10.2 Len homomorphism

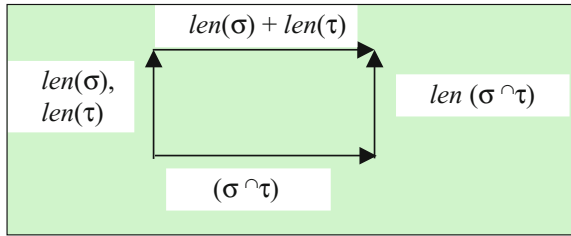
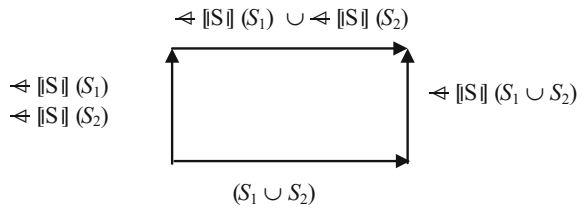


Fig. 10.3 Set removal endomorphism

$$\begin{aligned} \llbracket S \rrbracket (S_1 \cup S_2) &= \llbracket S \rrbracket (S_1) \cup \llbracket S \rrbracket (S_2) \\ \llbracket S \rrbracket (\emptyset) &= \emptyset \end{aligned}$$



Comment Monoids and their morphisms are useful and are widely used in VDM⁺. They are well-behaved structures and allow compact definitions of functions and also simplify proof. The use of monoids and morphisms helps to produce compact models and proofs.

10.3 Models and Modelling

A model is a mathematical representation of the physical world, and as it is a simplification of the reality, it does not include all aspects of the world. However, it is important that the model includes all essential aspects of the world.

The adequacy of a model is a key concern, and model exploration is a way to determine its adequacy. The model of the Tacoma Narrows Bridge² did not include aerodynamic forces, and this was a major factor in the eventual collapse of the bridge. Occasionally, there may be more than one model to explain the reality, and the decision then is to choose the “best” model. For example, Newtonian physics and the Theory of Relativity are both models of the physical world, and the latter is the appropriate model when dealing with velocities approaching the speed of light.

Occam’s Razor (or the “*Principle of Parsimony*”) is a key principle underlying modelling. It states “*Entia non sunt multiplicanda praeter necessitatem*”, which means that the number of entities required to explain anything should be kept to a minimum. That is, the modeller should seek the simplest model with the least number of assumptions, and all superfluous concepts that are not needed to explain the phenomena should be removed. The net result is a crisp and simpler model that captures the essence of the reality. The principle is attributed to the medieval philosopher William of Ockham.

The model is an abstraction or simplification of reality and serves as a way of testing hypotheses or ideas about some aspects of the world. This involves the formulation of questions which are then answered in terms of the model. Next, we consider sets, sequences, relations and functions.

10.4 Sets

Sets were discussed in Chap. 4, and this section focuses on their use in VDM^{*}. A set is a collection of objects all drawn from the same domain. Sets may be enumerated by listing all of their elements. Thus, the set of all even natural numbers less than or equal to 10 is:

$$\{2, 4, 6, 8, 10\}$$

The membership of a set S is denoted by $x \in S$. There is also another notation for set membership used in VDM^{*} based on the characteristic function.

$$\begin{aligned} \chi : \Sigma &\rightarrow \mathbb{P}\Sigma \rightarrow \mathbf{B} \\ \chi|[x] &S \underline{\Delta} x \in S \end{aligned}$$

The empty set is denoted by \emptyset . Various operations on sets such as union, intersection, difference and symmetric difference are employed. The union of two sets S and T is given by $S \cup T$ and their intersection by $S \cap T$. The set restriction operation for S on T restricts T to its elements that are in S and is given by:

²The Tacoma Narrows bridge (known as Galloping Gertie) collapsed in 1940 due to a design flaw. Further details are in [ORg:02].

$$\triangleleft[S]T = S \cap T$$

This is also written in infix form as:

$$S \triangleleft T = S \cap T$$

The set difference (or set removal operation) of two sets S , T is given by $S \setminus T$. It is also written as:

$$\triangleleft[T]S = S \setminus T$$

or in infix form as:

$$T \triangleleft S = S \setminus T$$

The symmetric difference operation is given by

$$S \Delta T \underline{\Delta} (S \cup T) \setminus (S \cap T)$$

The number of elements in a set S is given by the cardinality function $card(S)$.

$$card(S) = \#S = |S|$$

The powerset of a set X is the set of all subsets of X . It is denoted by $\mathbb{P}X$, and it includes the empty set. The notation $\mathbb{P}' Y$ denotes the set of non-empty subsets of Y , i.e. $\triangleleft[\emptyset] \mathbb{P}Y$.

The set S is said to be a subset of T ($S \subseteq T$) if whenever $s \in S$ then $s \in T$. The distributed union of set of sets is defined as:

$$\cup / \{S_1, S_2, \dots, S_n\} = S_1 \cup S_2 \cup \dots \cup S_n$$

10.5 Relations and Functions

There is no specific notation for relations in VDM⁺. Instead, relations from a set X to a set Y are modelled by either:

- $R \subseteq \mathbb{P}(X \times Y)$
- A partial functions ρ of the form $\rho: X \rightarrow \mathbb{P}' Y$.

An element x is related to y if:

- $(x, y) \in R$
- or
- $\chi[[x]] \rho \wedge y \in \rho(x)$

The structure $(X \rightarrow \mathbb{P}' Y)$ is isomorphic to $\mathbb{P}(X \times Y)$.

The functional view of relations uses the indexed monoid $(X \rightarrow \mathbb{P}' Y, \textcircled{+}, \theta)$, and this allows the familiar relational operations, such as relational inverse, relational composition, to be expressed functionally. For example, the inverse of a relation $\rho: (X \rightarrow \mathbb{P}' Y)$ is of the form $\rho^{-1}: (Y \rightarrow \mathbb{P}' X)$. The definition of the relational inverse is constructive.

A function takes values from one domain and returns values in another domain. The map $\mu: X \rightarrow Y$ denotes a partial function from the set X to the set Y . The result of the function for a particular value x in the domain of μ is given by $\mu(x)$. The empty map from X to Y is denoted by θ .

The domain of a map μ is given by $\text{dom } \mu$, and it gives the elements of X for which the map μ is defined. The notation $x \in \text{dom } \mu$ indicates that the element x is in the domain of μ . This is often written with a slight abuse of notation as $x \in \mu$. Clearly, $\text{dom } \theta = \emptyset$ and $\text{dom } \{x \rightarrow y\} = \{x\}$. The domain of μ is X if μ is total.

New maps may be constructed from existing maps using function override. The function override operator was defined in Z , and the operator combines two functions of the same type to give a new function of the same type. The effect of the override operation $(\mu \dagger v)$ is that an entry $\{x \mapsto y\}$ is removed from the map μ and replaced with the entry $\{x \mapsto z\}$ in v . The notation $(\mu \dagger v)$ is employed for function override in VDM^{*}.

$$\begin{aligned} (\mu \dagger v)(x) &= v(x) \text{ where } x \in \text{dom } v \\ (\mu \dagger v)(x) &= \mu(x) \text{ where } x \notin \text{dom } v \wedge x \in \text{dom } \mu \end{aligned}$$

Maps under override form a monoid $(X \rightarrow Y, \dagger, \theta)$ with the empty map θ the identity of the monoid. The domain (dom) operator is a monoid homomorphism. The domain homomorphism is of the form:

$$\begin{aligned} \text{dom} : (X \rightarrow Y, \dagger, \theta) &\rightarrow (\mathbb{P}X, \cup, \emptyset). \\ \text{dom } \{x \mapsto y\} &= \{x\} \end{aligned}$$

Domain removal and domain restriction operators were discussed for sets in the previous section. The domain removal operator ($\llbracket S \rrbracket$) and the domain restriction operator ($\langle \llbracket S \rrbracket \rangle$) are endomorphisms of $(X \rightarrow Y, \dagger, \theta)$.

The domain removal operator ($\llbracket S \rrbracket$) is defined as follows:

$$\begin{aligned} \llbracket S \rrbracket : (X \rightarrow Y, \dagger, \theta) &\rightarrow (X \rightarrow Y, \dagger, \theta) \\ \llbracket S \rrbracket \{x \mapsto y\} &\Delta \theta \quad (x \in S) \\ \llbracket S \rrbracket \{x \mapsto y\} &\Delta \{x \mapsto y\} \quad (x \notin S) \end{aligned}$$

The domain restriction operator ($\triangleleft[S]$) is defined as follows:

$$\begin{aligned} \triangleleft[S] &: (X \rightarrow Y, \dagger, \theta) \rightarrow (X \rightarrow Y, \dagger, \theta) \\ \triangleleft[S] \{x \mapsto y\} &\triangleq \{x \mapsto y\} \quad (x \in S) \\ \triangleleft[S] \{x \mapsto y\} &\triangleq \theta \quad (x \notin S) \end{aligned}$$

The restrict and removal operators are extended to restriction/removal from another map by abuse of notation:

$$\begin{aligned} \triangleleft[\mu]v &= \triangleleft[\text{dom } \mu]v \\ \triangleleft[\mu]v &= \triangleleft[\text{dom } \mu]v \end{aligned}$$

Given an *injective* total function $f : (X \rightarrow W)$ and a total function $g : (Y \rightarrow Z)$ then the map functor $(f \rightarrow g)$ is a homomorphism of

$$\begin{aligned} (f \rightarrow g) &: (X \rightarrow Y, \dagger, \theta) \rightarrow (W \rightarrow Z, \dagger, \theta) \\ (f \rightarrow g) \{x \mapsto y\} &= \{f(x) \mapsto g(y)\} \end{aligned}$$

Currying (named after the logician Haskell Curry) involves replacing a function of n arguments by the application of n functions of 1-argument. It is used extensively in VDM^{*}.

Consider the function $f : X \times Y \rightarrow Z$. Then, the usual function application is:

$$f(x, y) = z.$$

The curried form of the above is application is:

$$f : X \rightarrow Y \rightarrow Z$$

$f|[x]$ is a function: $Y \rightarrow Z$, and its application to y yields $z : f|[x]y = z$.

10.6 Sequences

Sequences are ordered lists of zero or more elements from the same set. The set of sequences from the set Σ is denoted by Σ^* , and the set of non-empty sequences is denoted by Σ^+ . Two sequences σ and τ are combined by sequence concatenation to give $\sigma \hat{\ } \tau$. The structure $(\Sigma^*, \hat{\ }, \Lambda)$ is a monoid under sequence concatenation, and the identity of the monoid is the empty sequence Λ .

The sequence constructor operator “:” takes an element x from the set Σ and a sequence σ from Σ^* , and produces a new sequence σ' that consists of the element x as the first element of σ' and the remainder of the sequence given by σ .

$$\sigma' = x : \sigma$$

The most basic sequence is given by:

$$\sigma = x : \Lambda$$

A sequence constructed of n elements $x_1, x_2, \dots, x_{n-1}, x_n$ (in that order) is given by:

$$x_1 : (x_2 : \dots : (x_{n-1} : (x_n : \Lambda)) \dots)$$

This is also written as:

$$\langle x_1, x_2, \dots, x_{n-1}, x_n \rangle$$

The head of a non-empty sequence is given by:

$$\begin{aligned} \text{hd} : \Sigma^+ &\rightarrow \Sigma \\ \text{hd}(x : \sigma) &= x \end{aligned}$$

The tail of a non-empty sequence is given by:

$$\begin{aligned} \text{tl} : \Sigma^+ &\rightarrow \Sigma^* \\ \text{tl}(x : \sigma) &= \sigma \end{aligned}$$

Clearly, for a non-empty sequence σ it follows that:

$$\text{hd}(\sigma) : \text{tl}(\sigma) = \sigma$$

The function *len* gives the length of a sequence (i.e. the number of elements in the sequence), and it is a monoid homomorphism from $(\Sigma^*, \cap, \Lambda)$ to $(\mathbb{N}, +, 0)$. The length of the empty sequence is clearly 0, i.e. $\text{len}(\Lambda) = 0$. The length of a sequence is also denoted by $|\sigma|$ or $\#\sigma$.

The elements of a sequence are given by the function *elems*. This is a monoid homomorphism from $(\Sigma^*, \cap, \Lambda)$ to $(\mathbb{P}\Sigma, \cup, \emptyset)$.

$$\begin{aligned} \text{elems} : \Sigma^* &\rightarrow \mathbb{P}\Sigma \\ \text{elems}(\Lambda) &= \emptyset \\ \text{elems}(x : \sigma) &= \{x\} \cup \text{elems}(\sigma) \end{aligned}$$

The elements of the empty sequence is the empty set \emptyset . The *elems* homomorphism loses information (e.g. the number of occurrences of each element in the sequence and the order in which the elements appear in the sequence). There is another operator (*items*) that determines the number of occurrences of each element in the sequence. The operator *items* generate a bag of elements from the sequence:

$$\text{items} : \Sigma^* \rightarrow (\Sigma \rightarrow \mathbb{N}_1).$$

The concatenation of two sequences is defined formally as:

$$\begin{aligned} \text{-}^\cap \text{-} : \Sigma^* \times \Sigma^* &\rightarrow \Sigma^* \\ \Lambda^\cap \sigma &= \sigma \\ (x : \sigma)^\cap \tau &= x : (\sigma^\cap \tau) \end{aligned}$$

The j th element in a sequence σ is given by $\sigma[i]$ where $1 \leq i \leq \text{len}(\sigma)$. The reversal of a sequence σ is given by $\text{rev } \sigma$.

10.7 Indexed Structures

An indexed monoid $(X \rightarrow M, \otimes, \theta)$ is created from an underlying base monoid $(M, *, u)$ and an index set X . It is defined as follows:

$$\begin{aligned} \otimes : (X \rightarrow M') \times (X \rightarrow M') &\rightarrow (X \rightarrow M') \\ \mu \otimes \theta &\quad \underline{\Delta} \mu \\ \mu \otimes (\{x \mapsto m\} \sqcup v) &\quad \underline{\Delta} (\mu \sqcup \{x \mapsto m\}) \otimes v \quad x \notin \mu \\ &\quad (\mu \uparrow \{x \mapsto \mu(x) * m\}) \otimes v \quad x \in \mu \wedge \mu(x) * m \neq u \\ &\quad \mu \otimes v \quad x \in \mu \wedge \mu(x) * m = u \end{aligned}$$

Indexing generates a higher monoid from the underlying base monoid, and this allows a chain (tower) of monoids to be built, with each new monoid built from the one directly underneath it in the chain. The power of the indexed monoid theorem is that it allows new structures to be built from existing structures, and the indexed structures inherit the properties of the underlying base structure.

A simple example of an indexed structure is a bag of elements from the set X . The indexed monoid is $(X \rightarrow \mathbb{N}_1, \oplus, \theta)$, and the underlying base monoid is $(\mathbb{N}, +, 0)$. Other indexed structures have also been considered in the Irish school of VDM.

10.8 Specifications and Proofs

Consider the specification of a simple dictionary in [But:00], where a dictionary is considered to be a set of words, and the dictionary is initially empty. There is an operation to insert a word into the dictionary, an operation to lookup a word in the dictionary, and an operation to delete a word from the dictionary.

$$\begin{aligned}
w &\in \text{Word} \\
\delta &: \text{Dict} = \mathbb{P} \text{Word} \\
\delta_0 &: \text{Dict} \\
\delta_0 &\underline{\Delta} \emptyset
\end{aligned}$$

The invariant is a condition (predicate expression) that is always true of the specification. The operations are required to preserve the invariant whenever the preconditions for the operations are true, and the initial system is required to satisfy the invariant. This gives rise to various proof obligations for the system.

The simple dictionary above is too simple for an invariant, but in order to illustrate the concepts involved, an artificial invariant that stipulates that all words in the dictionary are “British” English is considered part of the system.

$$\begin{aligned}
\text{isBritEng} &: \text{Word} \rightarrow \mathbf{B} \\
\text{inv-Dict} &: \text{Dict} \rightarrow \mathbf{B} \\
\text{inv-Dict } \delta &\underline{\Delta} \quad \forall [\text{isBritEng}] \delta
\end{aligned}$$

The signature of \forall is $(X \rightarrow \mathbf{B}) \rightarrow \mathbb{P} X \rightarrow \mathbf{B}$, and it is being used slightly differently from the predicate calculus. There is a proof obligation to show that the initial state of the dictionary (i.e., δ_0) satisfies the invariant. That is, it is required to show that $\text{inv-Dict } \delta_0 = \text{TRUE}$. However, this is clearly true since the dictionary is empty in the initial state.

The first operation considered is the operation to insert a word into the dictionary. The precondition to the operation is that the word is not currently in the dictionary and that the word is “British” English.

$$\begin{aligned}
\text{Ins} &: \text{Word} \rightarrow \text{Dict} \rightarrow \text{Dict} \\
\text{Ins} [w] \delta &\underline{\Delta} \delta \cup \{w\} \\
\text{pre-Ins} &: \text{Word} \rightarrow \text{Dict} \rightarrow \mathbf{B} \\
\text{pre-Ins} [w] \delta &\underline{\Delta} \text{isBritEng}(w) \wedge w \notin \delta
\end{aligned}$$

There is a proof obligation associated with the Ins operation. It states that if the invariant is true, and the precondition for the Ins operation is true, then the invariant is true following the Ins operation.

$$\text{inv-Dict } \delta \wedge \text{pre-Ins} [w] \delta \Rightarrow \text{inv-Dict}(\text{Ins} [w] \delta)$$

Comment One important difference between the Irish school of VDM and other methods such as classical VDM or Z is that postconditions are not employed in VDM^{*}. Instead, the operation is explicitly constructed.

Theorem

$$inv-Dict \delta \wedge pre-Ins|[w]| \delta \Rightarrow inv-Dict(Ins|[w]| \delta)$$

Proof

$$\begin{aligned} & inv-Dict \delta \wedge pre-Ins |[w]| \delta \\ \Rightarrow & \forall [isBritEng] \delta \wedge isBritEng(w) \wedge w \notin \delta \\ \Rightarrow & (\forall w_d \in \delta isBritEng(w_d)) \wedge isBritEng(w) \wedge w \notin \delta \\ \Rightarrow & \forall w_d \in (\delta \cup \{w\}) isBritEng(w_d) \\ \Rightarrow & \forall [isBritEng] (\delta \cup \{w\}) \\ \Rightarrow & inv-Dict (Ins|[w]| \delta) \end{aligned}$$

The next operation considered is a word lookup operation, and this operation returns true if the word is present in the dictionary and false otherwise. It is given by:

$$\begin{aligned} Lkp : & Word \rightarrow Dict \rightarrow \mathbf{B} \\ Lkp |[w]| \delta \underline{\Delta} & \chi |[w]| \delta \end{aligned}$$

The final operation considered is a word removal operation. This operation removes a particular word from the dictionary and is given by³:

$$\begin{aligned} Rem : & Word \rightarrow Dict \rightarrow Dict \\ Rem |[w]| \delta \underline{\Delta} & \ll |[w]| \delta \end{aligned}$$

There is a proof obligation associated with the Rem operation. It states that if the invariant is true, and the precondition for the Rem operation is true, then the invariant is true following the Rem operation.

$$inv-Dict \delta \wedge pre-Rem|[w]| \delta \Rightarrow inv-Dict(Rem|[w]| \delta)$$

10.9 Refinement in Irish VDM

A specification in the Irish school of VDM involves defining the state of the system and then specifying various operations. The formal specification is implemented by a programmer, and mathematical proof is employed to provide confidence that the program meets its specification. VDM^{*} employs many constructs that are not part

³Notation is often abused and this should strictly be written as $\ll|[w]| \delta$.

of conventional programming languages, and hence, there is a need to write an intermediate specification that is between the original specification and the eventual program code. The intermediate specification needs to be correct with respect to the specification, and the program needs to be correct with respect to the intermediate specification. This requires mathematical proof.

The representation of an abstract data type like a set by a sequence is termed data reification, and data reification is concerned with the process of transforming an abstract data type into a concrete data type. The abstract and concrete data types are related by the retrieve function, which maps the concrete data type to the abstract data type. There are typically several possible concrete data types for a particular abstract data type (i.e. refinement is a relation), whereas there is one abstract data type for a concrete data type (i.e. retrieval is a function). For example, sets are often reified to unique sequences, where several unique sequences can represent a set, whereas a unique sequence represents exactly one set.

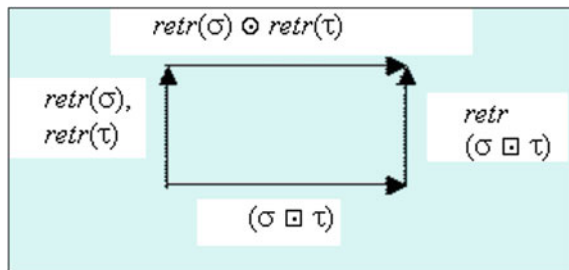
The operations defined on the concrete data type need to be related to the operations defined on the abstract data type. The commuting diagram property is required to hold; that is, for an operation \square on the concrete data type to correctly refine the operation \odot on the abstract data type, the following diagram must commute and the commuting diagram property (Fig. 10.4) requires proof. That is, it is required to prove that:

$$ret(\sigma \square \tau) = (ret \sigma) \odot (ret \tau)$$

It needs to be proved that the initial states correspond to one another, and that each operation in the concrete state is correct with respect to the operation in the abstract state, and also that it is applicable (i.e. whenever the abstract operation may be performed, then the concrete operation may be performed also).

The process of refinement of the dictionary from a set to a sequence of words is considered. This involves defining the concrete state and the operations on the state, and proving that the refinement is valid. The retrieve function derives the abstract state from the concrete state and is given by the elems operator for the set to sequence refinement of the dictionary. The following is adapted from [But:00]:

Fig. 10.4 Commuting diagram property



$$\begin{aligned}\sigma &\in DSeq = Word^* \\ \sigma_0 &: Dseq \\ \sigma_0 &\underline{\Delta} \Lambda\end{aligned}$$

$$inv-Dseq \underline{\Delta} \forall [[isBritEng]] \sigma$$

$$\begin{aligned}retr-Dict &: DSeq \rightarrow Dict \\ retr-Dict \sigma &\underline{\Delta} elems \sigma\end{aligned}$$

Here, \forall has signature $(X \rightarrow \mathbf{B}) \rightarrow X^* \rightarrow \mathbf{B}$.

The first operation considered on the concrete state is the operation to insert a word into the dictionary.

$$\begin{aligned}Ins_1 &: Word \rightarrow DSeq \rightarrow DSeq \\ Ins_1[[w]] \sigma &\underline{\Delta} w : \sigma\end{aligned}$$

$$\begin{aligned}pre-Ins_1 &: Word \rightarrow DSeq \rightarrow \mathbf{B} \\ pre-Ins_1[[w]] \sigma &\underline{\Delta} isBritEng(w) \wedge w \notin elems(\sigma)\end{aligned}$$

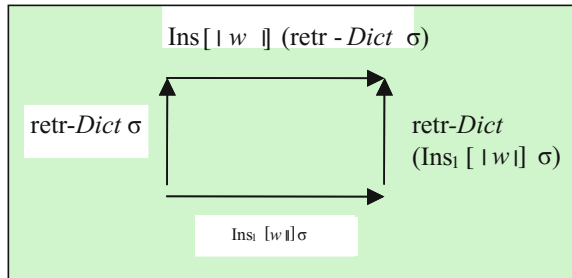
There is a proof obligation associated with the Ins_1 operation.

$$inv-DSeq \sigma \wedge pre-Ins_1[[w]] \sigma \Rightarrow inv-DSeq(Ins_1[[w]] \sigma)$$

The proof is similar to that considered earlier on the abstract state. Next, we show that Ins_1 is a valid refinement of Ins . This requires that the commuting diagram property holds (Fig. 10.5):

$$pre-Ins_1[[w]] \sigma \Rightarrow retr-Dict(Ins_1[[w]] \sigma) = Ins[[w]](retr-Dict \sigma)$$

Fig. 10.5 Commuting diagram for dictionary refinement



Proof

$$\begin{aligned}
 & pre\text{-Ins}_1|[w]| \sigma \\
 & \Rightarrow \text{isBritEng}(w) \wedge w \notin \text{elems}(\sigma) \\
 & \\
 & \text{retr-Dict}(\text{Ins}_1|[w]| \sigma) \\
 & = \text{retr-Dict}(w : \sigma) \\
 & = \text{elems}(w : \sigma) \\
 & = \{w\} \cup \text{elems}(\sigma) \\
 & = \{w\} \cup \text{retr-Dict}(\sigma) \\
 & = \text{Ins} |[w]| (\text{retr-Dict } \sigma)
 \end{aligned}$$

There are other operations for the concrete representation of the dictionary, and these are discussed in [But:00].

10.10 Review Questions

1. Describe how the Irish school of VDM differs from standard VDM.
2. Describe the various algebraic structures and their morphisms that are used in VDM⁺.
3. What is a model and explain the characteristics of a good model?
4. Explain the difference between a set and a sequence in VDM⁺.
5. Explain how relations are represented in the Irish school of VDM.
6. Describe the process of specification and refinement in VDM⁺.
7. Discuss the nature of proof in VDM⁺.

10.11 Summary

The Irish School of VDM is a variant of standard VDM and is characterized by its constructive approach, classical mathematical style and its terse notation. The method combines the “*what*” and “*how*” of formal methods in that its terse specification style stipulates in concise form *what* the system should do, and furthermore, the fact that its specifications are constructive (or functional) means that that the “*how*” is included with the “*what*”.

VDM^{*} follows a similar development methodology as in standard VDM and is a model-oriented approach. The initial specification is presented, with initial state and operations defined. The operations are presented with preconditions, and the operation is functionally constructed. Each operation has an associated proof obligation; if the precondition for the operation is true and the operation is performed, then the system invariant remains true after the operation.

The school has been influenced by the work of Polya and Lakatos. Polya has recommended problem solving by first tackling easier subproblems, whereas Lakatos adopted a heuristic approach to mathematical discovery based on proposing theorems and discovering hidden lemmas.

There is a rich operator calculus in the Irish school of VDM, and new operators and structures that are useful for specification and proof are sought. A special emphasis placed on the identification of useful structures and their morphisms that provide compact specifications and proof.

Partial functions are employed, and care is taken to ensure that the function is defined and will terminate prior to function application. The logic of partial functions (LPFs) is avoided, and care is taken to ensure that the recursive argument is strictly decreasing in each recursive invocation. The \perp symbol is typically used in the Irish school to represent *undefined or unavailable or do not care*. Academic and industrial projects have been conducted using VDM^{*}, but at this stage, tool support is limited.

The formal methods group at Trinity College, Dublin (www.cs.tcd.ie/fmg), is active in promoting the philosophy and method of the Irish school of VDM.

11.1 Introduction

The unified modelling language (UML) is a visual modelling language for software systems. It was developed by Jim Rumbaugh, Grady Booch and Ivar Jacobson [Jac:99a] at Rational Corporation (now part of IBM), as a notation for modelling object-oriented systems. It provides a visual means of specifying, constructing and documenting object-oriented systems, and it facilitates the understanding of the architecture of the system, and managing the complexity of a large system.

The language was strongly influenced by three existing methods: the *Object Modelling Technique* (OMT) developed by Rumbaugh; the *Booch Method* developed by Booch and *Object-Oriented Software Engineering* (OOSE) developed by Jacobson. UML unifies and improves upon these methods, and it has become a popular formal approach to modelling software systems.

Models provide a better understanding of the system to be developed, and a UML model allows the system to be visualized prior to its implementation. Large complex systems are difficult to understand in their entirety, and the use of a UML model is a way to simplify the underlying reality and to deal with complexity. The choice of the model is fundamental, and a good model will provide a good insight into the system. Models need to be explored and tested to ensure their adequacy as a representation of the system. Models simplify the reality, but it is important to ensure that the simplification does not exclude any important details. The chosen model affects the view of the system, and different roles require different viewpoints of the proposed system.

An architect will design a house prior to its construction, and the blueprints will contain details of the plan of each room, as well as plans for electricity and plumbing. That is, the plans for a house include floor plans, electrical plans and plumbing plans. These plans provide different viewpoints of the house to be constructed and are used to provide estimates of the time and materials required to construct it.

A database developer will often focus on entity-relationship models, whereas a systems analyst may often focus on algorithmic models. An object-oriented developer will focus on classes and on the interactions of classes. Often, there is a need to view the system at different levels of detail, and no single model in itself is sufficient for this. This leads to the development of a small number of interrelated models.

UML provides a formal model the system, and it allows the same information to be presented in several ways, and at different levels of detail. The requirements of the system are expressed in terms of use cases; the design view captures the problem space and solution space; the process view models the systems processes; the implementation view addresses the implementation of the system and the deployment view models the physical deployment of the system.

There are several UML diagrams providing different viewpoints of the system, and these provide the blueprint of the software. Next, we provide an overview of UML.

11.2 Overview of UML

UML is an expressive graphical modelling language for visualizing, specifying, constructing and documenting a software system. It provides several views of the software's architecture, and it has a clearly defined syntax and semantics. Each stakeholder (e.g. project manager, developers and testers) has a different perspective, and looks at the system in different ways at different times during the project. UML is a way to model the software system before implementing it in a programming language.

A UML specification consists of precise, complete and unambiguous models. The models may be employed to generate code in a programming language such as Java or C++. The reverse is also possible, and so it is possible to work with either the graphical notation of UML, or the textual notation of a programming language. UML expresses things that are best expressed graphically, whereas a programming language expresses things that are best expressed textually, and tools are employed to keep both views consistent. UML may be employed to document the software system, and it has been employed in several domains including the banking sector, defence and telecommunications.

The use of UML requires an understanding of its basic building blocks, the rules for combining the building blocks and the common mechanisms that apply throughout the language. There are three kinds of building blocks employed:

- Things;
- Relationships;
- Diagrams.

Things are the object-oriented building blocks of the UML. They include *structural things*, *behavioural things*, *grouping things* and *annotational things* (Table 11.1). Structural things are the nouns of the UML models; behavioural things are the dynamic parts and represent behaviour and their interactions over time; grouping things are the organization parts of UML and annotation things are the explanatory parts. Things, relationships and diagrams are all described graphically as discussed in [Jac:99a].

There are four kinds of relationship in UML:

- Dependency;
- Association;
- Generalization;
- Extensibility.

Dependency is used to represent a relationship between two elements of a system, in which a change to one thing affects the other thing (dependent thing). *Association* describes how elements in the UML diagram are associated and describes a set of connections among elements in a system. *Aggregation* is an association that represents a structural relationship between a whole and its parts.

Table 11.1 Classification of UML things

Thing	Kind	Description
Structural	Class	A class is a description of a set of objects that share the same attributes and operations
	Interface	An interface is a collection of operations that specify a service of a class or component. It specifies the externally visible behaviour
	Collaboration	A collaboration defines an interaction between software objects
	Use case	A use case is a set of actions that define the interaction between an actor and the system to achieve a particular goal
	Active class	An active class is used to describe concurrent behaviour of a system
	Component	A component is used to represent any part of a system for which UML diagrams are made
	Node	A node is used to represent a physical part of the system (e.g. server, network, etc.)
Behavioural	Interaction	These comprise interactions (message exchange between components) expressed as sequence diagrams or collaboration diagrams
	State machine	A state machine is used to describe different states of system components
Grouping	Packages	These are the organization parts of UML models. A package organizes elements into groups and is a way to organize a UML model
Annotation		These are the explanatory parts (notes) of UML

A *generalization* is a parent/child relationship in which the objects of the specialized element (child) are substituted for objects of the generalized element (the parent). *Extensibility* refers to a mechanism to extend the power of the language to represent extra behaviour of the system. Next, we describe the key UML diagrams.

11.3 UML Diagrams

The various UML diagrams provide a graphical visualization of the system from different viewpoints, and we present several key UML diagrams in Table 11.2.

The concept of class and objects are taken from object-oriented design, and classes are the most important building block of any object-oriented system. A class is a set of objects that share the same attributes, operations, relationships and semantics [Jac:99a]. Classes may represent software things and hardware things. For example, walls, doors, and windows are all classes, whereas individual doors and windows are objects. A class represents a set of objects rather than an individual object.

Automated bank teller machines (ATMs) include two key classes: customers and accounts. The class definition includes both the data structure for customers and accounts, and the operations on customers and accounts. These include operations to add or remove a customer, operations to debit or credit an account or to transfer from one account to another. There are several instances of customers and accounts, and these are the actual customers of the bank and their accounts.

Table 11.2 UML diagrams

Diagram	Description
Class	A class is a key building block of any object-oriented system. The class diagram shows the classes, their attributes and operations, and the relationships between them
Object	This shows a set of objects and their relationships. An object diagram is an instance of a class diagram
Use case	These show the actors in the system, and the different functions that they require from the system
Sequence	These diagrams show how objects interact with each other and the order in which the interactions occur
Collaboration	This is an interaction diagram that emphasizes the structural organization of objects that send and receive messages
State chart	These describe the behaviour of objects that act differently according to the state that they are in
Activity	This diagram is used to illustrate the flow of control in a system (it is similar to a flowchart)
Component	This diagram shows the structural relationship of components of a software system and their relationships/interfaces
Deployment	This diagram is used for visualizing the deployment view of a system and shows the hardware of the system and the software on the hardware

Table 11.3 Simple class diagram

Customer	Account
Name: String Address: String	Balance: Real Type: String
Add() Remove()	Debit() Credit() CheckBal() Transfer()

Every class has a name (e.g. Customer and Account) to distinguish it from other classes (Table 11.3). There will generally be several objects associated with the class. The class diagram describes the name of the class, its attributes and its operations. An attribute represents some property of the class that is shared by all objects; for example, the attributes of the class “Customer” are name and address. Attributes are listed below the class name, and the operations are listed below the attributes. The operations may be applied to any object in the class. The responsibilities of a class may also be included in the definition.

Class diagrams typically include various relationships between classes. In practice, very few classes are stand alone, and most collaborate with others in various ways. The relationship between classes needs to be considered, and these provide different ways of combining classes to form new classes. The relationships include dependencies (a change to one thing affects the dependent thing); generalizations (these link generalized classes to their specializations in a subclass/superclass relationship); and associations (these represent structural relationships among objects).

A dependency is a relationship that states that a change in the specification of one thing affects the dependent thing. It is indicated by a dashed line (—>). Generalizations allow a child class to be created from one or more parent classes (single or multiple inheritance). A class that has no parents is termed a base class (e.g. consider the base class Shape with three children: Rectangle, Circle and Polygon, and where Rectangle has one child namely Square). Generalization is indicated by a solid directed line that points to the parent (—►). Association is a structural relationship that specifies that objects of one thing are connected to objects of another thing. It is indicated by a solid line connecting the same or different classes.

The object diagram (Fig. 11.1) shows a set of objects and their relationships at a point of time. It is related to the class diagram in that the object is an instance of the class. The ATM example above has two classes (customers and accounts), and the objects of these classes are the actual customers and their corresponding accounts. Each customer may have several accounts, and the names and addresses of the customers are detailed as well as the corresponding balance in the customer’s accounts. There is one instance of the customer class and two instances of the account class in this example.

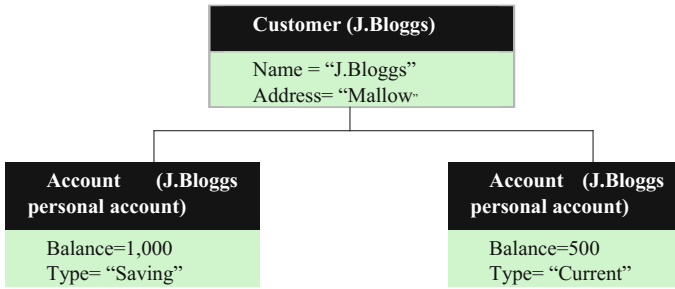


Fig. 11.1 Simple object diagram

An object has a state that has a given value at each time instance. Operations on the object will typically (with the exception of query operations) change its state. An object diagram contains objects and links to other objects and gives a snapshot of the system at a particular moment of time.

A use case diagram models the dynamic aspects of the system, and it shows a set of use cases and actors and their relationships. It describes scenarios (or sequences of actions) in the system from the user's viewpoint (actor) and shows how the actor interacts with the system. An actor represents the set of roles that a user can play, and the actor may be human or an automated system. Actors are connected to use cases by association, and they may communicate by sending and receiving messages.

A use case diagram shows a set of use cases, with each use case representing a functional requirement. Use cases are employed to model the visible services that the system provides within the context of its environment, and for specifying the requirements of the system as a black box. Each use case carries out some work that is of value to the actor, and the behaviour of the use case is described by the flow of events in text. The description includes the main flow of events for the use case and the exceptional flow of events. These flows may also be represented graphically. There may also be alternate flows as well as the main flow of the use case. Each sequence is termed a scenario, and a scenario is one instance of a use case.

Use cases provide a way for the end users and developers to share a common understanding of the system. They may be applied to all or part of the system (subsystem), and the use cases are the basis for development and testing. A use case is represented graphically by an ellipse. The benefits of use cases include:

- Enables the stakeholders (e.g. domain experts, developers, testers and end users) to share a common understanding of the functional requirements.
- Models the requirements (specifies what the system should do).
- Models the context of a system (identifies actors and their roles).
- May be used for development and testing.

Fig. 11.2 Use case diagram of ATM machine

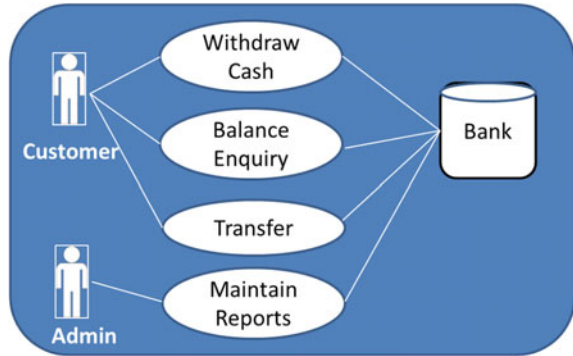


Figure 11.2 presents a simple example of the definition of the use cases for an ATM application. The typical user operations at an ATM machine include the balance inquiry operation, cash withdrawal and the transfer of funds from one account to another. The actors for the system include “customer” and “admin”, and these actors have different needs and expectations of the system.

The behaviour from the user’s viewpoint is described, and the use cases include “withdraw cash”, “balance enquiry”, “transfer” and “maintain/reports”. The use case view includes the actors who are performing the sequence of actions.

The next UML diagram considered is the sequence diagram which models the dynamic aspects of the system and shows the interaction between objects/classes in the system for each use case. The interactions model the flow of control that characterizes the behaviour of the system, and the objects that play a role in the interaction are identified. A sequence diagram emphasizes the time ordering of messages, and the interactions may include messages that are dispatched from object to object, with the messages ordered in sequence by time.

The example in Fig. 11.3 considers the sequences of interactions between objects for the “Balance Enquiry” use case. This sequence diagram is specific to the case of a valid balance enquiry, and a sequence diagram is needed to handle the exception cases as well.

The behaviour of the “balance enquiry” operation is evident from the diagram. The customer inserts the card into the ATM machine, and the PIN number is requested by the ATM. The customer then enters the number, and the ATM machine contacts the bank for verification of the number. The bank confirms the validity of the number and the customer then selects the balance enquiry operation. The ATM contacts the bank to request the balance of the particular account, and the bank sends the details to the ATM machine. The balance is displayed on the screen of the ATM machine. The customer then withdraws the card. The actual sequence of interactions is evident from the sequence diagram.

The example above has four objects (Customer, ATM, Bank and Account) and these are laid out from left to right at the top of the sequence diagram. Collaboration diagrams are interaction diagrams that consist of objects and their relationships. However, while sequence diagrams emphasize the time ordering of messages,

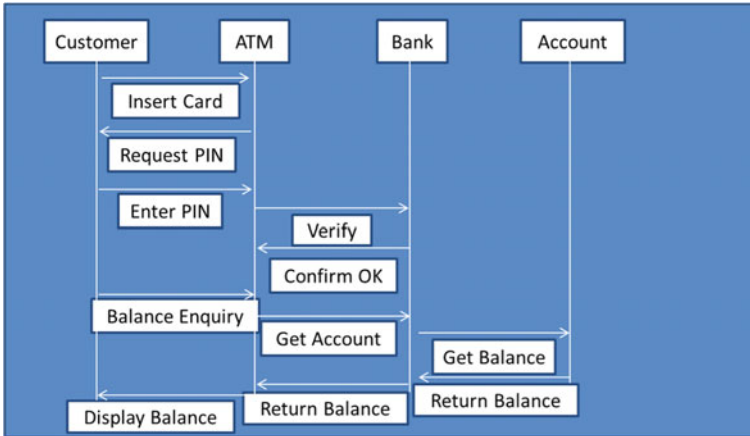


Fig. 11.3 UML sequence diagram for balance enquiry

a collaboration diagram emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams may be converted to the other without loss of information. Collaboration diagrams are described in more detail in [Jac:99a].

The activity diagram is considered in Fig. 11.4, and this is essentially a flowchart showing the flow of control from one activity to another. It is used to model the dynamic aspects of a system, and this involves modelling the sequential and possibly concurrent steps in a computational process. It is different from a sequence diagram in that it shows the flow from activity to activity, whereas a sequence diagram shows the flow from object to object.

State diagrams (also known as state machine diagrams or state charts) show the dynamic behaviour of a class, and how an object behaves differently depending on the state that it is in. There is an initial state and a final state, and the operation generally results in a change of state, with different states being entered and exited (Fig. 11.5). A state diagram is an enhanced version of a finite state machine (as discussed in Chap. 13).

There are several other UML diagrams including component and deployment diagrams. The reader is referred to [Jac:99a].

Advantages of UML

UML offers a rich notation to model software systems and to understand the proposed system from different viewpoints. Its main advantages are (Table 11.4).

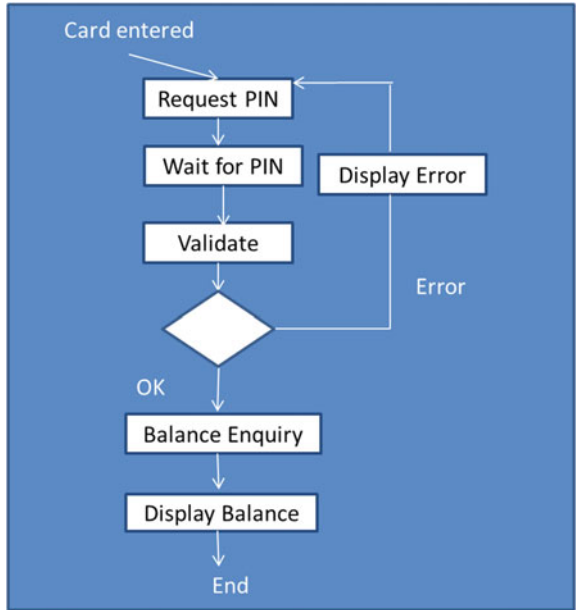


Fig. 11.4 UML activity diagram

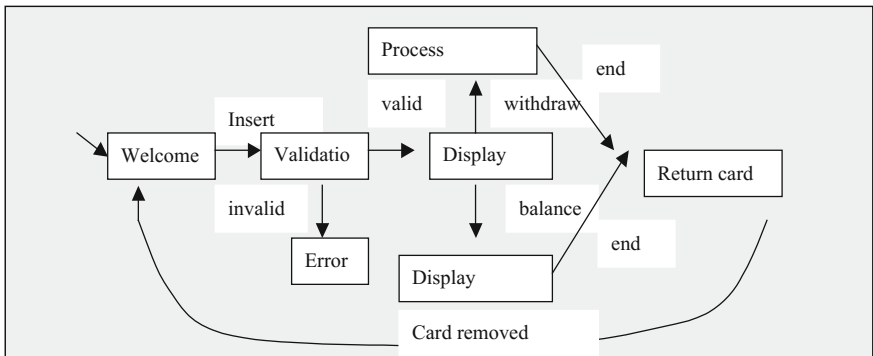


Fig. 11.5 UML state diagram

Table 11.4 Advantages of UML

Advantages of UML
Visual modelling language with a rich expressive notation
Mechanism to manage complexity of a large system
Enables the proposed system to be studied before implementation
Visualization of architecture design of the system
It provides different views of the system
Visualization of system from different viewpoints
Use cases allow the description of typical user behaviour
Better understanding of implications of user behaviour
Use cases provide a mechanism to communicate the proposed behaviour of the software system
Use cases are the basis of development and testing

11.4 Object Constraint Language

The object constraint language (OCL) is a declarative language that provides a precise way of describing rules (or expressing constraints) on the UML models. OCL was originally developed as a business modelling language by Jos Warmer at IBM, and it was developed further by the Object Management Group (OMG), as part of a formal specification language extension to UML. It was mainly used initially as part of UML, but it is now used independently of UML.

OCL is a pure expression language; i.e. there are no side effects as in imperative programming languages, and expressions can be used in various places in a UML model including:

- Specify the initial value of an attribute.
- Specify the body of an operation.
- Specify a condition.

There are several types of OCL constraints including (Table 11.5).

Table 11.5 OCL constraints

OCL constraint	Description
Invariant	A condition that must always be true. An invariant may be placed on an attribute in a class, and this has the effect of restricting the value of the attribute. All instances of the class are required to satisfy the invariant. An invariant is a predicate and is introduced after the keyword inv
Precondition	A condition that must be true before the operation is executed. A precondition is a predicate and is introduced after the keyword pre
Postcondition	A condition that must be true when the operation has just completed execution. A postcondition is a predicate and is introduced after the keyword post
Guard	A condition that must be true before the state transition occurs

Table 11.6 UML Tools

Tool	Description
Requisite Pro	Requirements and use case management tool. It provides requirements management and traceability
Rational Software Modeler (RSM)	Visual modelling and design tool that is used by systems architects/systems analysts to communicate processes, flows, and designs
Rational Software Architect (RSA)	RSA is a tool that enables good architectures to be created
Clearcase/Clearquest	These are configuration management/change control tools that are used to manage change in the project

There are various tools available to support OCL, and these include OCL compilers (or checkers) that provide syntax and consistency checking of the OCL constraints, and the USE specification environment is based on UML/OCL.

11.5 Industrial Tools for UML

Table 11.6 presents a small selection of the available tools that support UML. Tools to support formal methods are discussed in Chap. 17.

11.6 Rational Unified Process

Software projects need a well-structured development process to achieve their objectives. The *Rational Unified Development Software Process* (RUP) [Jac:99b] has become important, and RUP and UML are often used together. RUP is

- Use case driven;
- Architecture centric;
- Iterative and incremental.

It includes iterations, phases, workflows, risk mitigation, quality control, project management and configuration control. Software projects may be complex, and there are risks that requirements may be missed in the process, or that the interpretation of a requirement may differ between the customer and developer. RUP gathers requirements as use cases, which describe the functional requirements from the point of view of the users of the system.

The use case model describes what the system will do at a high-level, and there is user focus in defining the scope the project. Use cases drive the development process, and the developers create a series of design and implementation models that realize the use cases. The developers review each successive model for conformance to the use case model, and testing verifies that the implementation model correctly implements the use cases.

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the use cases and factors such as the platform that the software is to run on, deployment considerations, legacy systems and non-functional requirements.

A commercial software product is a large undertaking and the work is decomposed into smaller slices or mini-projects, where each mini-project is a manageable chunk. Each mini-project is an iteration that results in an increment to the product (Fig. 11.6).

Iterations refer to the steps in the workflow, and an increment leads to the growth of the product. If the developers need to repeat the iteration, then the organization loses only the misdirected effort of a single iteration, rather than the entire product. Therefore, the unified process is a way to reduce risk in software engineering. The early iterations implement the areas of greatest risk to the project.

RUP consists of four phases, and these are inception, elaboration, construction and transition (Fig. 11.7). Each phase consists of one or more iterations, and each iteration consists of several workflows. The workflows may be requirements,

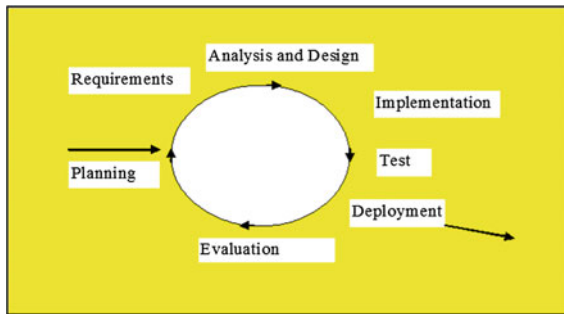


Fig. 11.6 Iteration in Rational Unified Process

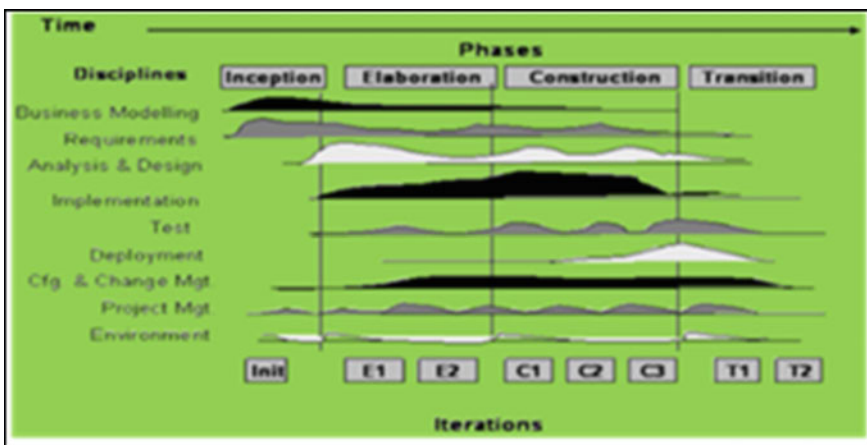


Fig. 11.7 Phases and workflows in Rational Unified Process

analysis, design, implementation and test. Each phase terminates in a milestone with one or more project deliverables.

The inception identifies and prioritizes the most important project risks, and it is concerned with initial project planning, cost estimation and early work on the architecture and functional requirements for the product. The elaboration phase specifies most of the use cases in detail. The construction phase is concerned with building the product and implements all agreed use cases. The transition phase covers the period during which the product moves into the customer site and includes activities such as training customer personnel, providing help-line assistance and correcting defects found after delivery.

The waterfall lifecycle has the disadvantage that the risk is greater towards the end of the project, where it is costly to undo mistakes from earlier phases. The iterative process develops an increment (i.e. a subset of the system functionality with the waterfall steps applied in the iteration), then another, and so on, and avoids developing the whole system in one step as in the waterfall methodology.

11.7 Review Questions

1. What is UML? Explain its main features.
2. Explain the difference between an object and a class.
3. Describe the various UML diagrams.
4. What are the advantages and disadvantages of UML?
5. What is the Rational Unified Process?
6. Describe the workflows in a typical iteration.
7. Describe the phases in the Rational Unified Process.

11.8 Summary

The unified modelling language is a visual modelling language for software systems, and it facilitates the understanding of the architecture, and management of the complexity of large systems. It was developed by Rumbaugh, Booch and Jacobson as a notation for modelling object-oriented systems, and it provides a visual means of specifying, constructing and documenting such systems. It facilitates the understanding of the architecture of the system and in managing its complexity.

UML allows the same information to be presented in several different ways and at different levels of detail. The requirements of the system are expressed in use cases; and other views include the design view that captures the problem space and solution space; the process view which models the systems processes; the implementation view and the deployment view.

The UML diagrams provide different viewpoints of the system, and provide the blueprint of the software. These include class and object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, activity diagrams, state charts, collaboration diagrams and deployment diagrams.

RUP consists of four phases, and these are inception, elaboration, construction and transition. Each phase consists of one or more iterations, and the iteration consists of several workflows. The workflows may be requirements, analysis, design, implementation and test. Each phase terminates in a milestone with one or more project deliverables.

12.1 Introduction

Edsger W. Dijkstra, C.A.R. Hoare and David Parnas are famous names in computer science, and they have received numerous awards for their contribution to the discipline. Their work has provided a scientific basis for computer software development and a rigorous approach to the development of software. We present a selection of their contributions in this chapter, including Dijkstra's calculus of weakest preconditions; Hoare's axiomatic semantics and Parnas's tabular expressions. There is more detailed information on the contributions of these pioneers in [ORg:06].

Mathematics and Computer Science were regarded as two completely separate disciplines in the 1960s, and software development was based on the assumption that the completed code would always contain defects. It was therefore better and more productive to write the code as quickly as possible and to then perform debugging to find the defects. Programmers then corrected the defects, made patches and retested and found more defects. This continued until they could no longer find defects. Of course, there was always the danger that defects remained in the code that could give rise to software failures.

John McCarthy argued at the IFIP congress in 1962 that the focus should instead be to prove that the programs have the desired properties, rather than testing the program *ad nauseum*. Robert Floyd believed that there was a way to construct a rigorous proof of the correctness of the programs using mathematics, and he demonstrated techniques (based on assertions) in a famous paper in 1967 that mathematics could be used for program verification. The NATO conference on software engineering in 1968 highlighted the extent of the problems that existed with software, and the term "*software crisis*" was coined to describe this. The problems included cost and schedule overruns and problems with the reliability of the software.

Fig. 12.1 Edsger Dijkstra.
Courtesy of Brian Randell



Dijkstra (Fig. 12.1) was born in Rotterdam in Holland, and he studied mathematics and physics at the University of Leyden. He obtained a PhD in Computer Science from the University of Amsterdam in 1959. He decided not to become a theoretical physicist, as he believed that programming offered a greater intellectual challenge.

He commenced his programming career at the Mathematics Centre in Amsterdam in the early 1950s, and he invented the shortest path algorithm in the mid-1950s. He contributed to the definition of Algol 60, and he designed and coded the first Algol 60 compiler.

Dijkstra has made many contributions to computer science, including contributions to language development, operating systems, formal program development and to the vocabulary of Computer Science. He received the Turing award in 1972, and some of his achievements are listed in Table 12.1.

Dijkstra advocated simplicity, precision and mathematical integrity in his formal approach to program development. He insisted that programs should be composed correctly using mathematical techniques and not debugged into correctness. He considered testing to be an inappropriate means of building quality into software, and his statement on software testing is well known:

Table 12.1 Dijkstra's achievements

Area	Description
Go to statement	Dijkstra argued against the use of the goto statement in programming. This eventually led to its abolition in programming
Graph algorithms	Dijkstra developed several efficient graph algorithms to determine the <i>shortest</i> or <i>longest</i> paths from a vertex u to vertex v in a graph
Operating systems	Dijkstra introduced ideas such as semaphores and deadly embrace, and that operating systems can be built as synchronized sequential processes
Algol 60	Dijkstra contributed to the definition of the language, and he designed and coded the first Algol 60 compiler
Formal program development (guarded commands and predicate transformers)	Dijkstra introduced guarded commands and predicate transformers as a means of defining the semantics of a programming language. He showed how weakest preconditions can be used as a calculus (<i>wp</i> -calculus) to develop reliable programs. This led to a science of programming using mathematical logic as a methodology for formal program construction His approach involves the development of programs from mathematical axioms

Testing a program shows that it contains errors never that it is correct.¹

Dijkstra corresponded with other academics through an informal distribution network known as the EWD series. These contain his various personal papers including trip reports and technical papers.

Charles Anthony Richard (C.A.R or Tony) Hoare studied philosophy (including Latin and Greek) at Oxford University (Fig. 12.2). He studied Russian at the Royal Navy during his National Service in the late 1950s. He then studied statistics and went to Moscow University as a graduate student to study machine translation of languages and probability theory. He discovered the well-known sorting algorithm “*Quicksort*”, while investigating efficient ways to look up words in a dictionary.

He returned to England in 1960 and worked as a programmer for Elliot Brothers (a company that manufactured scientific computers). He led a team to produce the first commercial compiler for Algol 60, and it was a very successful project. He then led a team to implement an operating system, and the project was a disaster. He managed a recovery from the disaster and then moved into the research division of the company.

¹Software testing is an essential part of the software process, and various types of testing are described in [ORg:02]. Modern software testing is quite rigorous and can provide a high degree of confidence that the software is fit for use. It cannot, of course, build quality in; rather, it can provide confidence that quality has been built in. The analysis of the defects identified during testing may be useful in improving the software development process.

Fig. 12.2 C.A.R Hoare

He took a position at Queens University in Belfast in 1968, and his research goals included examining techniques to assist with the implementation of operating systems, especially to see if advances in programming methodologies could assist with the problems of concurrency. He also published material on the use of assertions to prove program correctness.

He moved to Oxford University in 1977 following the death of Christopher Strachey (well known for his work in denotational semantics) and built up the programming research group. This group later developed the Z specification language and CSP, and Hoare received the ACM Turing award in 1980. Following his retirement from Oxford, he took up a position as senior researcher at Microsoft Research in the UK.

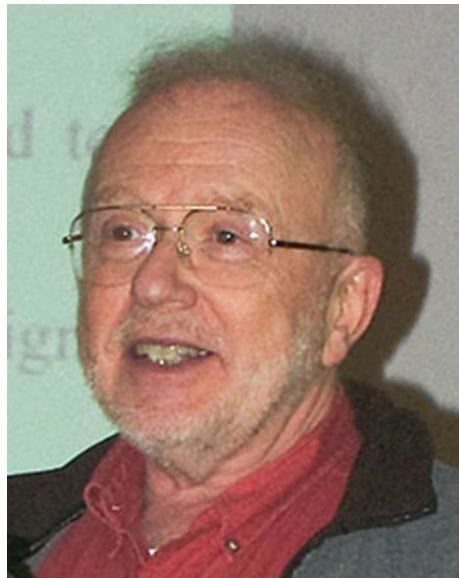
Hoare has made many contributions to computer science and these include the quicksort algorithm, the axiomatic approach to program semantics, and programming constructs for concurrency (Table 12.2). He remarked on the direction of the Algol programming language:

Algol 60 was a great achievement in that it was a significant advance over most of its successors.

Hoare has made fundamental contributions to programming languages, and his 1980 ACM Lecture on the “*Emperors Old Clothes*” is well known. He stresses the importance of communicating ideas (as well as having ideas) and enjoys writing (and rewriting).

Table 12.2 Hoare's achievements

Area	Description
Quicksort	Quicksort is a highly efficient sorting algorithm
Axiomatic semantics	Hoare defined a small programming language in terms of axioms and logical inference rules for proving partial correctness of programs
Communicating Sequential Processes (CSP)	CSP is a mathematical approach to the study of communication and concurrency. It is applicable to the specification and design of computer systems that continuously interact with their environment

Fig. 12.3 David Parnas

David L. Parnas (Fig. 12.3) has been influential in the computing field, and his ideas on the specification, design, implementation remain important. He has won numerous awards (including ACM best paper award in 1979); influential paper awards from ICSE; the ACM SigSoft outstanding researcher award and honorary doctorates for his contribution to Computer Science.

He studied at Carnegie Mellon University and was awarded B.S., M.S., and PhD degrees in Electrical Engineering by the university. He has worked in both industry and academia, and his approach aims to achieve a middle way between theory and practice. His research has focused on real industrial problems that engineers face and on finding solutions to these practical problems. Several organizations such as Phillips in the Netherlands; the Naval Research Laboratory (NRL) in Washington; IBM Federal Systems Division and the Atomic Energy Board of Canada have benefited from his advice and expertise.

He advocates a solid engineering approach to the development of high-quality software and argues that software engineers² today do not have the right engineering education to perform their roles effectively. The role of engineers is to apply scientific principles and mathematics to design and develop useful products. He argues that the level of mathematics taught in most Computer Science courses is significantly less than that taught to traditional engineers. In fact, computer science graduates often enter the work place with knowledge of the latest popular technologies, but with only a limited knowledge of the foundations needed to be successful in producing safe and useful products. Consequently, he argues that it should not be surprising that the quality of software produced today falls below the desired standard, as the current approach to software development is informal and based on intuition rather than sound engineering principles. He argues that computer scientists should be educated as engineers and provided with the right scientific and mathematical background to do their work effectively.

Parnas has made a strong contribution to software engineering, including contributions to requirements specification, software design, software inspections, testing, tabular expressions, predicate logic and ethics for software engineers (Table 12.3). His reflections on software engineering remain valuable and contain the insight gained over a long career.

12.2 Calculus of Weakest Preconditions

The weakest precondition calculus was developed by Dijkstra [Dij:76] and applied to the formal development of programs. This section is based on material from [Gri:81], and a programming notation is introduced and defined in terms of the weakest precondition. The weakest precondition $wp(S, R)$ is a predicate that describes a set of states, and it is a function with two arguments that results in a predicate. The function has two arguments (a command and a predicate), where the predicate argument describes the set of states satisfying R after the execution of the command. It is defined as follows:

Definition (Weakest Precondition)

The predicate $wp(S, R)$ represents the set of all states such that, if execution of S commences in any one of them, then it is guaranteed to terminate in a state satisfying R .

Let S be the assignment command $i := i + 5$, and let R be $i \leq 3$ then

$$wp(i := i + 5; i \leq 3) = (i \leq -2)$$

²Parnas argues that the term ‘engineer’ should be used only in its classical sense as a person who is qualified and educated in science and mathematics to design and inspect products. The evolution of language that has led to a debasement of the term ‘engineer’ with various groups who do not have the appropriate background to be considered ‘engineers’ in the classical sense applying this title.

Table 12.3 Parnas's achievements

Area	Description
Tabular expressions	Tabular expressions are mathematical tables that are employed for specifying requirements. They enable complex predicate logic expressions to be represented in a simpler form
Mathematical documentation	He advocates the use of mathematical documents for software engineering that are precise and complete. These documents are for system requirements, system design, software requirements, module interface specification and module internal design
Requirements specification	His approach to requirements specification (developed with Kathryn Heninger and others) involves the use of mathematical relations to specify the requirements precisely
Software design	His contribution to software design was revolutionary. A module is characterized by its knowledge of a design decision (secret) that it hides from all others. This is known as the information hiding principle, and it allows software to be designed for changeability. Every information-hiding module has an interface that provides the only means to access the services provided by the modules. The interface hides the module's implementation. Information hiding is used in object-oriented programming
Software inspections	His approach to software inspections is quite distinct from the well-known Fagan inspection methodology. The reviewers are required to take an active part in the inspection and are provided with a list of questions by the author. The reviewers are required to provide documentation of their analysis to justify the answers to the individual questions. This involves the production of mathematical tables
Predicate logic	He introduced a novel approach to deal with undefined values ^a in predicate logic expressions which preserves the two-valued logic. His approach is quite distinct from the logic of partial functions developed by Cliff Jones [Jon:86]
Industry contributions	His industrial contribution is impressive including work on defining the requirements of the A7 aircraft and the inspection of safety critical software for the automated shutdown of the nuclear power plant at Darlington
Ethics for software engineers	He has argued that software engineers have a professional responsibility to build safe products, to accept individual responsibility for their design decisions, and to be honest about current software engineering capabilities. He applied these principles in arguing against the strategic defence initiative (SDI) of the Reagan administration in the mid 1980s

^aHis approach allows undefinedness to be addressed in predicate calculus while maintaining the two-valued logic. A primitive predicate logic expression that contains an undefined term is considered false in the calculus, and this avoids the three-valued logics developed by Jones and Dijkstra

The weakest precondition $wp(S, T)$ represents the set of all states such that if execution of S commences in any one of them, then it is guaranteed to terminate.

$$wp(i := i + 5; T) = T$$

The weakest precondition $wp(S, R)$ is a precondition of S with respect to R , and it is also the weakest such precondition. Given another precondition P of S with respect to R , then $P \Rightarrow wp(S, R)$.

For a fixed command S then $wp(S, R)$ can be written as a function of one argument: $wp_S(R)$, and the function wp_S transforms the predicate R to another predicate $wp_S(R)$. In other words, the function wp_S acts as a *predicate transformer*.

An imperative program may be regarded as a predicate transformer. This is since a predicate P characterizes the set of states in which the predicate P is true, and an imperative program may be regarded as a binary relation on states, leading to the Hoare triple $P\{F\}Q$. That is, the program F acts as a predicate transformer. The predicate P may be regarded as an input assertion, i.e. a predicate that must be true before the program F is executed. The predicate Q is the output assertion, and is true if the program F terminates, having commenced in a state satisfying P .

12.2.1 Properties of WP

The weakest precondition $wp(S, R)$ has several well-behaved properties as described in Table 12.4.

Table 12.4 Properties of WP

Property	Description
Law of the excluded miracle $wp(S, F) = F$	This describes the set of states such that if execution commences in one of them, then it is guaranteed to terminate in a state satisfying false. However, no state ever satisfies false, and therefore $wp(S, F) = F$. The name of this law derives from the fact that it would be a miracle if execution could terminate in no state
Distributivity of conjunction $wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R)$	This property stipulates that the set of states such that if execution commences in one of them, then it is guaranteed to terminate in a state satisfying $Q \wedge R$ is precisely the set of states such that if execution commences in one of them then execution terminates with both Q and R satisfied
Law of monotonicity $Q \Rightarrow R$ then $wp(S, Q) \Rightarrow wp(S, R)$	This property states that if a postcondition Q is stronger than a postcondition R , then the weakest precondition of S with respect to Q is stronger than the weakest precondition of S with respect to R
Distributivity of disjunction $wp(S, Q) \vee wp(S, R) \Rightarrow wp(S, Q \vee R)$	This property states that the set of states corresponding to the weakest precondition of S with respect to Q or the set of states corresponding to the weakest precondition of S with respect to R is stronger than the weakest precondition of S with respect to $Q \vee R$. Equality holds for distributivity of disjunction only when the execution of the command is deterministic

12.2.2 WP of Commands

The weakest precondition can be used to provide the definition of commands in a programming language. The commands considered are taken from [Gri:81].

- **Skip Command**

$$wp(skip, R) = R$$

The *skip* command does nothing and is used to explicitly say that nothing should be done. The predicate transformer wp_{skip} is the identity function.

- **Abort Command**

$$wp(abort, R) = F$$

The *abort* command is executed in a state satisfying false (i.e. no state). This command should never be executed. If program execution reaches a point where *abort* is to be executed then the program is in error and abortion is called for.

- **Sequential Composition**

$$wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$$

The sequential composition command composes two commands S_1 and S_2 by first executing S_1 and then executing S_2 . Sequential composition is expressed by $S_1; S_2$.

Sequential composition is associative:

$$wp(S_1; (S_2; S_3), R) = wp((S_1; S_2); S_3, R)$$

- **Simple Assignment Command**

$$wp(x := e, R) = dom(e) \text{ \textbf{and} } R_e^x$$

The execution of the *assignment* command consists of evaluating the value of the expression e and storing its value in the variable x . However, the command may be executed only in a state where e may be evaluated.

The expression R_e^x denotes the expression obtained by substituting e for all free occurrences of x in R . For example,

$$(x + y > 2)_v^x = v + y > 2$$

The **cand** operator is used to deal with undefined values, and it was discussed in Chap. 7. It is a non-commutative operator and the expression $a \mathbf{cand} b$ is equivalent to:

$$a \mathbf{cand} b \cong \mathbf{if} \ a \ \mathbf{then} \ b \ \mathbf{else} \ F$$

The explanation of the definition of the weakest precondition of the assignment statement $wp(x := e, R)$ is that R will be true after execution if and only if the predicate R with the value of x replaced by e is true before execution (since x will contain the value of e after execution).

Often, the domain predicate $dom(e)$ that describes the set of states that e may be evaluated is omitted as assignments are usually written in a context in which the expressions are defined.

$$wp(x := e, R) = R_e^x$$

The simple assignment can be extended to a multiple assignment to simple variables. The assignment is of the form $x_1, x_2, \dots, x_n := e_1, e_2, \dots, e_n$ and is described in [Gri:81].

- **Assignment to Array Element Command**

$$wp(b[i] := e, R) = \mathit{inrange}(b, i) \mathbf{cand} \mathit{dom}(e) \mathbf{cand} R_{(b;i:e)}^b$$

The execution of the *assignment to an array element* command consists of evaluating the expression e and storing its value in the array element subscripted by i . The $\mathit{inrange}(b, i)$ and $\mathit{dom}(e)$ are usually omitted in practice as assignments are usually written in a context in which the expressions are defined and the subscripts are in range. Therefore, the weakest precondition is given by:

$$wp(b[i] := e, R) = R_{(b;i:e)}^b$$

The notation $(b;i:e)$ denotes an array identical to array b except that the array element subscripted by i contains the value e . The explanation of the definition of the weakest precondition of the assignment statement to an array element

($wp(b[i] := e, R)$) is that R will be true after execution if and only if the value of b replaced by $(b;i:e)$ is true before execution (since b will become $(b;i:e)$ after execution).

- **Alternate Command**

$$wp(IF, R) = \text{dom}(B_1 \vee B_2 \vee \dots \vee B_n) \wedge (B_1 \vee B_2 \vee \dots \vee B_n) \\ \wedge (B_1 \Rightarrow wp(S_1, R)) \wedge (B_2 \Rightarrow wp(S_2, R)) \wedge \dots \wedge (B_n \Rightarrow wp(S_n, R))$$

The *alternate* command is the familiar **if** statement of programming languages. The general form of the alternate command is:

```

if   $B_1 \rightarrow S_1$ 
       $\square$   $B_2 \rightarrow S_2$ 
       $\dots$ 
       $\square$   $B_n \rightarrow S_n$ 
fi

```

Each $B_i \rightarrow S_i$ is a guarded command (S_i is any command). The guards must be well defined in the state where execution begins, and at least one of the guards must be true or execution aborts. If at least one guard is true, then one guarded command $B_i \rightarrow S_i$ with true guard B_i is chosen and S_i is executed.

For example, in the if statement below, the statement $z := x + 1$ is executed if $x > 2$, and the statement $z := x + 2$ is executed if $x < 2$. For $x = 2$ either (but not both) statements are executed. This is an example of non-determinism.

```

if  $x \geq 2 \rightarrow z := x + 1$ 
       $\square$   $x \leq 2 \rightarrow z := x + 2$ 
fi

```

- **Iterative Command**

The *iterate* command is the familiar while loop statement of programming languages. The general form of the iterate command is:

```

do   $B_1 \rightarrow S_1$ 
       $\square$   $B_1 \rightarrow S_1$ 
       $\dots$ 
       $\square$   $B_n \rightarrow S_n$ 
od

```

The meaning of the iterate command is that a guard B_i is chosen that is true, and the corresponding command S_i is executed. The process is repeated until there are no more true guards. Each choice of a guard and execution of the corresponding statement is an iteration of the loop. On termination of the iteration command all of the guards are false.

The meaning of the DO command $wp(DO, R)$ is the set of states in which execution of DO terminates in a bounded number of iterations with R true.

$$wp(DO, R) = (\exists k : 0 \leq k : H_k(R))$$

where $H_k(R)$ is defined as:

$$H_k(R) = H_0(R) \vee wp(IF, H_{k-1}(R))$$

A more detailed explanation of loops is in [Gri:81]. The definition of procedure call may be given in weakest preconditions also.

12.2.3 Formal Program Development with WP

The use of weakest preconditions for formal program development is described in [Gri:81]. The approach is a radical departure from current software engineering, and it involves developing the program and a formal proof of its correctness together. A program P is correct with respect to a precondition Q and a postcondition R if $\{Q\}P\{R\}$, and the idea is that the program and its proof should be developed together. The proof involves weakest preconditions and uses the formal definition of the programming constructs (e.g. assignment and iteration) as discussed earlier.

Programming is viewed as a goal-oriented activity in that the desired result (i.e. the postcondition R) plays a more important role in the development of the program than the precondition Q . Programming is employed to solve a problem, and the problem needs to be clearly stated with precise preconditions and postconditions.

The example of a program³ P to determine the maximum of two integers x and y is discussed in [Gri:81]. A program P is required that satisfies:

$$\{T\}P\{R : z = \max(x, y)\}$$

The postcondition R is then refined by replacing \max with its definition:

$$\{R : (z \geq x \wedge z \geq y) \wedge (z = x \vee z = y)\}$$

³Many of these examples are considered “toy programs” when compared to real-world industrial software development, but they illustrate the concepts involved in developing software rigorously using the weakest precondition calculus.

The next step is to identify a command that could be executed in order to establish the postcondition R . One possibility is $z := x$ and the conditions under which this assignment establishes R is given by:

$$\begin{aligned} \text{wp}(z := x, R) &= x \geq x \wedge x \geq y \wedge (x = x \vee x = y) \\ &= x \geq y \end{aligned}$$

Another possibility is $z := y$ and the conditions under which this assignment establishes R is given by:

$$\text{wp}(z := y, R) = y \geq x$$

The desired program is then given by:

```

if  $x \geq y \rightarrow z := x$ 
 $\square$   $y \geq x \rightarrow z := y$ 
fi

```

There are many more examples of formal program development in [Gri:81].

12.3 Axiomatic Definition of Programming Languages

An assertion is a property of the program's objects: e.g. the assertion $(x - y > 5)$ is an assertion that may or may not be satisfied by a state of the program during execution. For example, the state in which the values of the variables x and y are 7 and 1, respectively, satisfies the assertion; whereas a state in which x and y have values 4 and 2, respectively, does not.

Robert Floyd (Fig. 12.4) did pioneering work on software engineering from the 1960s, including important contributions to the theory of parsing; the semantics of programming languages and methodologies for the creation of efficient and reliable software.

Floyd believed that there was a way to construct a rigorous proof of the correctness of the programs using mathematics. He showed that mathematics could be used for program verification, and he introduced the concept of *assertions* that provided a way to verify the correctness of programs. His first article on program proving techniques based on assertions was in 1967 [Flo:67].

Floyd's 1967 paper was concerned with assigning meaning to programs, and he also introduced the idea of a loop invariant. His approach was based on programs expressed by flowcharts, and an assertion was attached to the edge of the flowchart. The meaning was that the assertion would be true during execution of the corresponding program whenever execution reached that edge. For a loop, Floyd placed an assertion P on a fixed position of the cycle, and proved that if execution

Fig. 12.4 Robert Floyd

commenced at the fixed position with P true, and reached the fixed position again, then P would still be true.

Flowcharts were employed in the 1960s to explain the sequence of basic steps for computer programs. Floyd's insight was to build upon flowcharts and to apply *an invariant assertion to each branch* in the flowchart. These assertions state the essential relations that exist between the variables at that point in the flowchart. An example relation is " $R = Z > 0, X = 1, Y = 0$ ". He devised a general flowchart language to apply his method to programming languages. The language essentially contains boxes linked by flow of control arrows.

Consider the assertion Q that is true on entry to a branch where the condition at the branch is P . Then, the assertion on exit from the branch is $Q \wedge \neg P$ if P is false and $Q \wedge P$ otherwise (Fig. 12.5).

The use of assertions may be employed in an assignment statement. Suppose x represents a variable and v represents a vector consisting of all the variables in the program. Suppose $f(x, v)$ represents a function or expression of x and the other program variables represented by the vector v . Suppose the assertion $S(f(x, v), v)$ is true before the assignment $x = f(x, v)$. Then the assertion $S(x, v)$ is true after the assignment (Fig. 12.6). This is given by:

Floyd used flowchart symbols to represent entry and exit to the flowchart. This included entry and exit assertions to describe the program's entry and exit conditions.

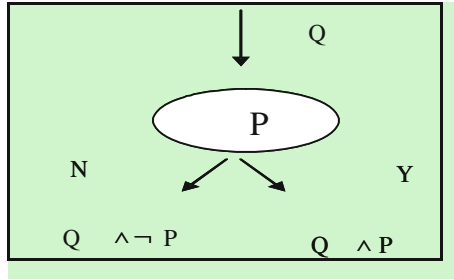


Fig. 12.5 Branch assertions in flowcharts

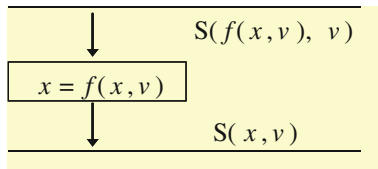


Fig. 12.6 Assignment assertions in flowcharts

Floyd’s technique showed how a computer program is a sequence of logical assertions. Each assertion is true whenever control passes to it, and statements appear between the assertions. The initial assertion states the conditions that must be true for execution of the program to take place, and the exit assertion essentially describes what must be true when the program terminates.

He recognized that if it can be shown that the assertion immediately following each step is a consequence of the assertion immediately preceding it, then the assertion at the end of the program will be true, provided the appropriate assertion was true at the beginning of the program.

His influential 1967 paper, “*Assigning Meanings to Programs*” influenced Hoare’s work on preconditions and postconditions leading to Hoare logic [Hor:69]. Hoare recognized that Floyd’s approach provided an effective method for proving the correctness of programs, and he built upon Floyd’s work to cover the familiar constructs of high-level programming languages. Floyd’s paper also presented a formal grammar for flowcharts, together with rigorous methods for verifying the effects of basic actions like assignments.

Hoare logic is a formal system of logic for programming semantics and program verification, and it was originally published in Hoare’s 1969 paper “*An axiomatic basis for computer programming*” [Hor:69]. Hoare and others have subsequently refined it, and it provides a logical methodology for precise reasoning about the correctness of computer programs. The well-formed formulae of the logical system are of the form:

$$P\{a\}Q$$

where P is the precondition; a is the program fragment and Q is the postcondition. The precondition P is a predicate (or input assertion), and the postcondition R is a predicate (output assertion). The braces separate the assertions from the program fragment. The well-formed formula $P\{a\}Q$ is itself a predicate that is either true or false. This notation expresses the partial correctness of a with respect to P and Q , where *partial correctness* and *total correctness* are defined as follows:

Definition (Partial Correctness)

A program fragment a is partially correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied and the execution terminates, then the resulting state satisfies Q .

The proof of partial correctness requires proof that the postcondition Q is satisfied if the program terminates. Partial correctness is a useless property unless termination is proved, as any non-terminating program is partially correct with respect to any specification.

Definition (Total Correctness)

A program fragment a is totally correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied then execution terminates and the resulting state satisfies Q .

The proof of total correctness requires proof that the postcondition Q is satisfied and that the program terminates. Total correctness is expressed by $\{P\} a \{Q\}$. The calculus of weakest preconditions developed by Dijkstra (discussed in the previous section) is based on total correctness, whereas Hoare's approach is based on partial correctness.

Hoare's axiomatic theory of programming languages consists of axioms and rules of inference to derive certain pre-post formulae. The meaning of several constructs in programming languages is presented here in terms of pre-post semantics.

- **Skip**

The meaning of the skip command is:

$$P\{skip\}P$$

Skip does nothing and it's this instruction guarantees that whatever condition is true on entry to the command is true on exit from the command.

- **Assignment**

The meaning of the assignment statement is given by the axiom:

$$P_e^x\{x := e\}P$$

The notation P_e^x has been discussed previously and denotes the expression obtained by substituting e for all free occurrences of x in P .

The meaning of the assignment statement is that P will be true after execution if and only if the predicate P_e^x with the value of x replaced by e in P is true before execution (since x will contain the value of e after execution).

- **Compound**

The meaning of the conditional command is:

$$\frac{P\{S_1\}Q, Q\{S_2\}R}{P\{S_1; S_2\}R}$$

The execution of the **compound** statement involves the execution of S_1 followed by S_2 . The correctness of the compound statement with respect to P and R is established by proving that the correctness of S_1 with respect to P and Q , and the correctness of S_2 with respect to Q and R .

- **Conditional**

The meaning of the conditional command is:

$$\frac{P \wedge B\{S_1\}Q, P \wedge \neg B\{S_2\}Q}{P\{\mathbf{if } B \mathbf{ then } S_1 \mathbf{ else } S_2\}Q}$$

The execution of the **if** statement involves the execution of S_1 or S_2 . The execution of S_1 takes place only when B is true, and the execution of S_2 takes place only when $\neg B$ is true. The correctness of the **if** statement with respect to P and Q is established by proving that S_1 and S_2 are correct with respect to P and Q .

However, S_1 is executed only when B is true, and therefore it is required to prove the correctness of S_1 with respect to $P \wedge B$ and Q , and the correctness of S_2 with respect to $P \wedge \neg B$ and Q .

- **While Loop**

The meaning of the while loop is given by:

$$\frac{P \wedge B\{S\}P}{P\{\mathbf{while } B \mathbf{ do } S\}P \wedge \neg B}$$

The property P is termed the loop invariant as it remains true throughout the execution of the loop. The invariant is satisfied before the loop begins and each iterations of the loop preserves the invariant.

The execution of the **while loop** is such that if the truth of P is maintained by one execution of S , then it is maintained by any number of executions of S . The execution of S takes place only when B is true, and upon termination of the loop $P \wedge \neg B$ is true.

Loops may fail to terminate and therefore there is a need to prove termination. The loop invariant needs to be determined for formal program development.

12.4 Tabular Expressions

Tables of constants have been used for millennia to define mathematical functions. The tables allow the data to be presented in an organized form that is easy to reference and use. The data presented in tables is well-organized and provides an explicit definition of a mathematical function. This allows the computation of the function for a particular value to be easily done. The use of tables is prevalent in schools where primary school children are taught multiplication tables and high school students refer to sine or cosine tables. The invention of electronic calculators may lead to a reduction in the use of tables as students may compute the values of functions immediately.

Tabular expressions are a generalization of tables in which constants may be replaced by more general mathematical expressions. Conventional mathematical expressions are a special case of tabular expressions. In fact, everything that can be expressed as a tabular expression can be represented by a conventional expression. Tabular expressions can represent sets, relations, functions and predicates and conventional expressions. A tabular expression may be represented by a conventional expression, but its advantage is that the tabular expression is easier to read and use, since a complex conventional expression is replaced by a set of simpler expressions.

Tabular expressions are invaluable in defining a piecewise continuous function, as it is relatively easy to demonstrate that all cases have been considered in the definition. It is easy to miss a case or to give an inconsistent definition in the conventional definition of a piecewise continuous function. The evaluation of a tabular expression is easy once the type of tabular expression is known. Tabular expressions have been applied to practical problems including the precise documentation of the system requirements of the A7 aircraft [Par:01].

Tabular expressions have been applied to precisely document the system requirements and to solve practical industrial problems. A collection of tabular expressions are employed to document the system requirements. The meaning of these tabular expressions in terms of their component expressions was done by Parnas [Par:92]. He identified several types of tabular expressions and provided a formal meaning for each type. A more general model of tabular expressions was proposed by Janicki [Jan:97], although this approach was based on diagrams using an artificial cell connection graph to explain the meaning of the tabular expressions. Parnas and others have proposed a general mathematical foundation for tabular expressions.

The function $f(x, y)$ is defined in the tabular expression below. The tabular expressions consist of headers and a main grid. The headers define the domain of the function and the main grid gives the definition. It is easy to see that the function is defined for all values on its domain as the headers are complete. It is also easy to see that the definition is consistent as the headers partition the domain of the function.

The evaluation of the function for a particular value (x, y) involves determining the appropriate row and column from the headers of the table and computing the grid element for that row and column (Fig. 12.7).

For example, the evaluation of $f(2, 3)$ involves the selection of row 1 of the grid (as $x = 2 \geq 0$ in H_1) and the selection of column 3 (as $y = 3 < 5$ in H_2). Hence, the value of $f(2, 3)$ is given by the expression in row 1 and column 3 of the grid, i.e. $-y^2$ evaluated with $y = 3$ resulting in -9 . The table simplifies the definition of the function. Tabular expressions have several applications (Table 12.5).

Examples of Tabular Expressions

The objective of this section is to illustrate the power of tabular expressions by considering a number of examples. The more general definition of tabular expressions allows for multidimensional tables, including multiple headers, and supports rectangular and non-rectangular tables. However, the examples presented here will be limited to two-dimensional rectangular tables, and will usually include two headers and one grid, with the meaning of the tables given informally.

The role of the headers and grid will become clearer in the examples, and usually, the headers contain predicate expressions, whereas the grid usually contains terms. However, the role of the grid and the headers change depending on the type of table being considered.

Normal Function Table

The first table that we discuss is termed the normal function table, and this table consists of two headers (H_1 and H_2) and one grid G . The headers are predicate expressions that partition the domain of the function; header H_1 partitions the domain of y , whereas header H_2 partitions the domain of x . The grid consists of terms. The function $f(x, y)$ is defined by the following table (Fig. 12.8):

		H ₂		
		$y = 5$	$y > 5$	$y < 5$
H ₁	$x \geq 0$	0	y^2	$-y^2$
	$x < 0$	x	$x+y$	$x-y$
		G		

Fig. 12.7 Tabular expressions (normal table)

Table 12.5 Applications of tabular expressions

Applications of tabular expressions
Specify requirements
Specify module interface design
Description of implementation of module
Mathematical software inspections

		H ₂		
		$x < 0$	$x = 0$	$x > 0$
H ₁	$y < 0$	$x^2 - y^2$	$x^2 - y^2$	$x^2 + y^2$
	$y = 0$	$x - y$	$x + y$	$x + y$
	$y > 0$	$x + y$	$x + y$	$x^2 + y^2$

Fig. 12.8 Normal table

The evaluation of the function $f(x, y)$ for a particular value of x, y is given by:

1. Determine the row i in header H₁ that is true.
2. Determine the column j in header H₂ that is true.
3. The evaluation of $f(x, y)$ is given by $G(i, j)$.

For example, the evaluation of $f(-2, 5)$ involves row 3 of H₁ as y is 5 (>0) and column 1 of header H₂ as x is -2 (<0). Hence, the element in row 3 and column 1 of the grid is selected (i.e. the element $x + y$). The evaluation of $f(-2, 5)$ is $-2 + 5 = 3$.

The usual definition of the function $f(x, y)$ defined piecewise is:

$$\begin{aligned}
 f(x, y) &= x^2 - y^2 && \text{where } x \leq 0 \wedge y < 0; \\
 f(x, y) &= x^2 + y^2 && \text{where } x > 0 \wedge y < 0; \\
 f(x, y) &= x + y && \text{where } x \geq 0 \wedge y = 0; \\
 f(x, y) &= x - y && \text{where } x < 0 \wedge y = 0; \\
 f(x, y) &= x + y && \text{where } x \leq 0 \wedge y > 0; \\
 f(x, y) &= x^2 + y^2 && \text{where } x > 0 \wedge y > 0;
 \end{aligned}$$

The danger with the usual definition of the piecewise function is that it is more difficult to be sure that every case has been considered, as it is easy to miss a case or for the cases to be overlap. Care needs to be taken with the value of the function on the boundary, as it is easy to introduce inconsistencies. It is straightforward to check that the tabular expression has covered all cases, and that there are no overlapping cases. This is done by examination of the headers to check for consistency and completeness. The headers for the tabular representation of $f(x, y)$ must partition the values that x and y may take, and this is clear from an examination of the headers.

Normal relation tables and predicate expression tables are interpreted similarly to normal function tables except that the grid entries are predicate expressions rather than terms as in the normal function table. The result of the evaluation of a predicate expression table is a Boolean value of true or false, whereas the result of the evaluation of the normal relation table is a relation. A characteristic predicate table is similar except that it is interpreted as a relation whose domain and range consist of tuples of fixed length. Each element of the tuple is a variable and the tuples are of the form $((x_1, x_2, \dots, x_n), (x_1', x_2', \dots, x_n'))$.

Inverted Function Table

The inverted function table is different from the normal table in that the grid contains predicates, and the header H₂ contains terms. The function $f(x, y)$ is defined by the following inverted table (Fig. 12.9):

		$x + y$	$x - y$	xy	H_2
		$x < 0$	$x = 0$	$x > 0$	
H_1	$y < 0$	$x < 0$	$x = 0$	$x > 0$	G
	$y = 0$	$x > 0$	$x < 0$	$x = 0$	
	$y > 0$	$x = 0$	$x < 0$	$x > 0$	

Fig. 12.9 Inverted table

The evaluation of the function $f(x, y)$ for a particular value of x, y is given by:

1. Determine the row i in header H_1 that is true.
2. Select row i of the grid and determine the column j of row i that is true.
3. The evaluation of $f(x, y)$ is given by $H_2(j)$.

For example, the evaluation of $f(-2, 5)$ involves the selection of row 3 of H_1 as y is 5 (>0). This means that row 3 of the grid is then examined and as x is -2 (<0) column 2 of the grid is selected. Hence, the element in column 2 of H_2 is selected as the evaluation of $f(x, y)$ (i.e. the element $x - y$). The evaluation of $f(-2, 5)$ is therefore $-2 - 5 = -7$.

The usual definition of the function $f(x, y)$ defined piecewise is:

$$\begin{aligned}
 f(x, y) &= x + y && \text{where } x < 0 \wedge y < 0; \\
 f(x, y) &= x - y && \text{where } x = 0 \wedge y < 0; \\
 f(x, y) &= xy && \text{where } x > 0 \wedge y < 0; \\
 f(x, y) &= x + y && \text{where } x > 0 \wedge y = 0; \\
 f(x, y) &= x - y && \text{where } x < 0 \wedge y = 0; \\
 f(x, y) &= xy && \text{where } x = 0 \wedge y = 0; \\
 f(x, y) &= x + y && \text{where } x = 0 \wedge y > 0; \\
 f(x, y) &= x - y && \text{where } x < 0 \wedge y > 0; \\
 f(x, y) &= xy && \text{where } x > 0 \wedge y > 0;
 \end{aligned}$$

Clearly, the tabular expression provides a more concise representation of the function. The inverted table arises naturally when there are many cases to consider, but only a few distinct values of the function. The function $f(x, y)$ can also be represented in an equivalent normal function table. In fact, any function that can be represented by an inverted function table may be represented in a normal function table and vice versa.

Inverted predicate expression tables and inverted relation tables are interpreted similarly to inverted function tables except that the header H_1 consists of predicate expressions rather than terms. The result of the evaluation of an inverted predicate expression table is the Boolean value true or false, whereas the evaluation of an inverted relation table is a relation.

There is more detailed information on Parnas’s contributions to software engineering, including software requirements, software design and software inspections in [Par:01].

12.5 Review Questions

1. What are Dijkstra's main achievements in computer science?
2. Describe Dijkstra's weakest precondition calculus and its application to formal program development.
3. What are Hoare's main achievements in computer science?
4. Describe Hoare's axiomatic semantics and its application to the correctness of computer programs.
5. What are Parnas's main achievements in computer science?
6. Describe tabular expressions and their applications.
7. What is a normal function table? What is an inverted function table?
8. Investigate Floyd's contributions to the computing field.

12.6 Summary

Dijkstra, Hoare and Parnas have made important contributions to computer science, and they have received numerous awards in recognition of their achievements. Their work has provided a scientific basis for computer software development, and we presented a selection of their contributions in this chapter.

Dijkstra has made contributions to language development, operating systems, formal program development and to the vocabulary of Computer Science. His calculus of weakest preconditions is used for the formal development of computer programs, where a program and its proof of correctness are developed together.

Hoare has developed the quicksort algorithm, the axiomatic approach to program semantics, and programming constructs for concurrency. He was responsible for producing the first commercial compiler for Algol 60 at Elliot Brothers.

Parnas has made a strong contribution to software engineering, including contributions to requirements specification, software design, software inspections, testing, tabular expressions, predicate logic and ethics for software engineers. His reflections on software engineering remain valuable and contain the insight gained over a long career. His tabular expressions are useful in defining piecewise continuous functions, where tabular expressions are a generalization of tables in which constants can be replaced by more general mathematical expressions.

13.1 Introduction

Automata theory is the branch of computer science that is concerned with the study of abstract machines and automata. These include finite-state machines, pushdown automata and Turing machines. Finite-state machines are abstract machines that may be in one of a finite number of states. These machines are in only one state at a time (current state), and the input symbol causes a transition from the current state to the next state. Finite-state machines have limited computational power due to memory and state constraints, but they have been applied to a number of fields including communication protocols, neurological systems and linguistics.

Pushdown automata have greater computational power than finite-state machines, and they contain extra memory in the form of a stack from which symbols may be pushed or popped. The state transition is determined from the current state of the machine, the input symbol and the element on the top of the stack. The action may be to change the state and/or push/pop an element from the stack.

The Turing machine is the most powerful model for computation, and this theoretical machine is equivalent to an actual computer in the sense that it can compute exactly the same set of functions. The memory of the Turing machine is a tape that consists of a potentially infinite number of one-dimensional cells. It provides a mathematical abstraction of computer execution and storage, as well as providing a mathematical definition of an algorithm. However, Turing machines are not suitable for programming, and therefore they do not provide a good basis for studying programming and programming languages.

13.2 Finite-State Machines

Warren McCulloch and Walter Pitts (two neurophysiologists) published early work on finite-state automata in 1943. They were interested in modelling the thought process for humans and machines. Moore and Mealy developed this work further in the mid-1950s, and their finite-state machines are referred to as the “*Mealy machine*” and the “*Moore machine*”. The Mealy machine determines its outputs from the current state and the input, whereas the output of Moore’s machine is based upon the current state alone.

Definition 13.1 (*Finite-State Machine*) A finite-state machine (FSM) is an abstract mathematical machine that consists of a finite number of states. It includes a start state q_0 in which the machine is in initially; a finite set of states Q ; an input alphabet Σ ; a state transition function δ and a set of final accepting states F (where $F \subseteq Q$).

The state transition function δ takes the current state and an input symbol and returns the next state. That is, the transition function is of the form:

$$\delta : Q \times \Sigma \rightarrow Q$$

The transition function provides rules that define the action of the machine for each input symbol, and its definition may be extended to provide output as well as a transition of the state. State diagrams are used to represent finite-state machines, and each state accepts a finite number of inputs. A finite-state machine (Fig. 13.1) may be deterministic or non-deterministic, and a *deterministic machine* changes to exactly (or at most)¹ one state for each input transition, whereas a *non-deterministic machine* may have a choice of states to move to for a particular input symbol.

Finite-state automata can compute only very primitive functions, and so they are not adequate as a model for computing. There are more powerful automata such as the Turing machine that is essentially a finite automaton with a potentially infinite storage (memory). Anything that is computable is computable by a Turing machine.

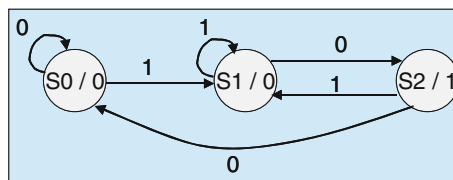


Fig. 13.1 Finite-state machine with output

¹The transition function may be undefined for a particular input symbol and state.

A finite-state machine can model a system that has a finite number of states, and a finite number of inputs/events that trigger transitions between states. The behaviour of the system at a point in time is determined from its current state and input, with behaviour defined for the possible input to that state. The system starts in an initial state.

A finite-state machine (also known as finite-state automata) is a quintuple $(\Sigma, Q, \delta, q_0, F)$. The alphabet of the FSM is given by Σ ; the set of states is given by Q ; the transition function is defined by $\delta : Q \times \Sigma \rightarrow Q$; the initial state is given by q_0 and the set of accepting states is given by F (where F is a subset of Q). A string is given by a sequence of alphabet symbols; that is, $s \in \Sigma^*$, and the transition function δ can be extended to $\delta^* : Q \times \Sigma^* \rightarrow Q$.

A string $s \in \Sigma^*$ is accepted by the finite-state machine if $\delta^*(q_0, s) = q_f$ where $q_f \in F$, and the set of all strings accepted by a finite-state machine is the language generated by the machine. A finite-state machine is termed *deterministic* (Fig. 13.2) if the transition function δ is a function,² otherwise (where it is a relation) it is said to be *non-deterministic*. A non-deterministic automaton is one for which the next state is not uniquely determined from the present state and input symbol, and the transition may be to a set of states rather than to a single state.

For the example above, the input alphabet is given by $\Sigma = \{0, 1\}$; the set of states by $\{A, B, C\}$; the start state by A; the accepting states by $\{C\}$ and the transition function is given by the state transition table below (Table 13.1). The language accepted by the automata is the set of all binary strings that end with a one that contains exactly two ones.

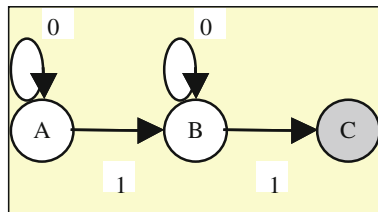


Fig. 13.2 Deterministic FSM

Table 13.1 State transition table

State	0	1
A	A	B
B	B	C
C	–	–

²It may be a total or a partial function (as discussed in Chap. 4).

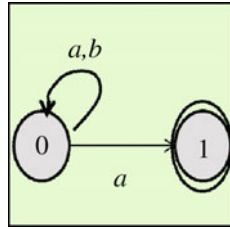


Fig. 13.3 Non-deterministic finite-state machine

A *non-deterministic* automaton (NFA) or non-deterministic finite-state machine is a finite-state machine where from each state of the machine and any given input, the machine may go to several possible next states. However, a non-deterministic automaton (Fig. 13.3) is equivalent to a deterministic automaton, in that they both recognize the same formal language (i.e. regular languages as defined in Chomsky's classification). For any non-deterministic automaton, it is possible to construct the equivalent deterministic automaton using power set construction.

NFAs were introduced by Scott and Rabin in 1959, and a NFA is defined formally as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ as in the definition of a deterministic automaton, and the only difference is in the transition function δ .

$$\delta : Q \times \Sigma \rightarrow \mathbb{P}Q$$

The non-deterministic finite-state machine $M_1 = (Q, \Sigma, \delta, q_0, F)$ may be converted to the equivalent deterministic machine $M_2 = (Q', \Sigma, \delta', q_0', F')$ where:

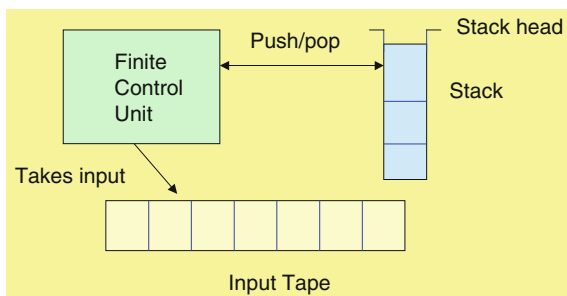
$$\begin{aligned} Q' &= \mathbb{P}Q \text{ (the set of all subsets of } Q) \\ q_0' &= \{q_0\} \\ F' &= \{q \in Q' \text{ and } q \cap F \neq \emptyset\} \\ \delta'(q, \sigma) &= \bigcup_{p \in q} \delta(p, \sigma) \text{ for each state } q \in Q' \text{ and } \sigma \in \Sigma. \end{aligned}$$

The set of strings (or language) accepted by an automaton M is denoted $L(M)$. That is, $L(M) = \{s \mid \delta^*(q_0, s) = q_f \text{ for some } q_f \in F\}$. A language is termed regular if it is accepted by some finite-state machine. Regular sets are closed under union, intersection, concatenation, complement and transitive closure. That is, for regular sets $A, B \subseteq \Sigma^*$ then:

- $A \cup B$ and $A \cap B$ are regular.
- $\Sigma^* \setminus A$ (i.e. A^c) is regular.
- AB and A^* is regular.

The proof of these properties is demonstrated by constructing finite-state machines to accept these languages. The proof for $A \cap B$ is to construct a machine $M_A \cap B$ that mimics the execution of M_A and M_B and is in a final state if and only if both M_A and M_B are in a final state. Finite-state machines are useful in designing systems that process sequences of data.

Fig. 13.4 Components of pushdown automata



13.3 Pushdown Automata

A pushdown automaton (PDA) is essentially a finite-state machine with a stack, and its three components (Fig. 13.4) are an input tape; a control unit and a potentially infinite stack. The stack head scans the top symbol of the stack, and two operations (push or pop) may be performed on the stack. The *push* operation adds a new symbol to the top of the stack, whereas the *pop* operation reads and removes an element from the top of the stack.

A pushdown automaton may remember a potentially infinite amount of information, whereas a finite-state automaton remembers only a finite amount of information. A PDA also differs from a FSM in that it may use the top of the stack to decide on which transition to take, and it may manipulate the stack as part of performing a transition. The input and current state determine the transition of a finite-state machine, and the FSM has no stack to work with.

A pushdown automaton is defined formally as a 7-tuple $(\Sigma, Q, \Gamma, \delta, q_0, Z, F)$. The set Σ is a finite set which is called the input alphabet; the set Q is a finite set of states; Γ is the set of stack symbols; δ is the transition function which maps $Q \times \{\Sigma \cup \{\varepsilon\}\}^3 \times \Gamma$ into finite subsets of $Q \times \Gamma^*$; q_0 is the initial state; Z is the initial stack top symbol on the stack (i.e. $Z \in \Gamma$) and F is the set of accepting states (i.e. $F \subseteq Q$).

Figure 13.5 shows a transition from state q_1 to q_2 , which is labelled as $a, b \rightarrow c$. This means that if the input symbol a occurs in state q_1 , and the symbol on the top of the stack is b , then b is popped from the stack and c is pushed onto the stack. The new state is then q_2 .

In general, a pushdown automaton has several transitions for a given input symbol, and so pushdown automata are mainly *non-deterministic*. If a pushdown automaton has at most one transition for the same combination of state, input symbol, and top of stack symbol, it is said to be a *deterministic PDA* (DPDA). The set of strings (or language) accepted by a pushdown automaton M is denoted $L(M)$.

³The use of $\{\Sigma \cup \{\varepsilon\}\}$ is to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

⁴This could also be written as $\delta : Q \times \{\Sigma \cup \{\varepsilon\}\} \times \Gamma \rightarrow \mathbb{P}(Q \times \Gamma^*)$. It may also be described as a transition relation.

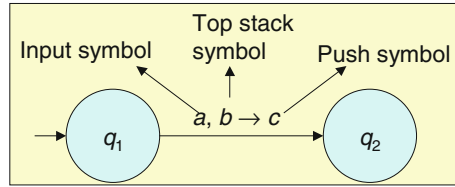


Fig. 13.5 Transition in pushdown automata

The class of languages accepted by pushdown automata is the context-free languages, and every context-free grammar can be transformed into an equivalent non-deterministic pushdown automaton. There is more detailed information on the classification of languages in Chap. 12 of [ORg:16b].

Example (Pushdown Automata) Construct a non-deterministic pushdown automaton which recognizes the language $\{0^n 1^n \mid n \geq 0\}$.

Solution We construct a pushdown automaton $M = (\Sigma, Q, \Gamma, \delta, q_0, Z, F)$ where $\Sigma = \{0, 1\}$; $Q = \{q_0, q_1, q_f\}$; $\Gamma = \{A, Z\}$; q_0 is the start state; the start stack symbol is Z and the set of accepting states is given by $\{q_f\}$. The transition function (relation) δ is defined by

1. $(q_0, 0, Z) \rightarrow (q_0, AZ)$
2. $(q_0, 0, A) \rightarrow (q_0, AA)$
3. $(q_0, \epsilon, Z) \rightarrow (q_1, Z)$
4. $(q_0, \epsilon, A) \rightarrow (q_1, A)$
5. $(q_1, 1, A) \rightarrow (q_1, \epsilon)$
6. $(q_1, \epsilon, Z) \rightarrow (q_f, Z)$

The transition function (Fig. 13.6) essentially says that whenever the value 0 occurs in state q_0 , then A is pushed onto the stack. Parts (3) and (4) of the transition function essentially state that the automaton may move from state q_0 to state q_1 at any moment. Part (5) states when the input symbol is 1 in state q_1 , then one symbol A is popped from the stack. Finally, part (6) states the automaton may move from state q_1 to the accepting state q_f only when the stack consists of the single stack symbol Z.

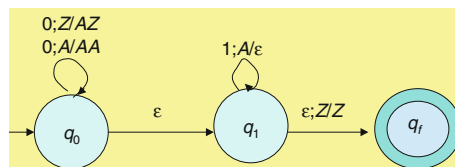


Fig. 13.6 Transition function for pushdown automata M

For example, it is easy to see that the string 0011 is accepted by the automaton, and the sequence of transitions is given by:

$$(q_0, 0011, Z) \vdash (q_0, 011, AZ) \vdash (q_0, 11, AAZ) \vdash (q_1, 11, AAZ) \vdash (q_1, 1, AZ) \vdash (q_1, \varepsilon, Z) \vdash (q_f, Z).$$

13.4 Turing Machines

Turing introduced the theoretical Turing machine in 1936, and this abstract mathematical machine consists of a head and a potentially infinite tape that is divided into frames (Fig. 13.7). Each frame may be either blank or printed with a symbol from a finite alphabet of symbols. The input tape may initially be blank or have a finite number of frames containing symbols. At any step, the head can read the contents of a frame; the head may erase a symbol on the tape, leave it unchanged or replace it with another symbol. It may then move one position to the right, one position to the left or not at all. If the frame is blank, the head can either leave the frame blank or print one of the symbols.

Turing believed that a human with finite equipment and with an unlimited supply of paper to write on could do every calculation. The unlimited supply of paper is formalized in the Turing machine by a paper tape marked off in squares, and the tape is potentially infinite in both directions. The tape may be used for intermediate calculations as well as input and output. The finite number of configurations of the Turing machine was intended to represent the finite states of mind of a human calculator.

The transition function determines for each state and the tape symbol what the next state to move to and what should be written on the tape, and where to move the tape head. The Turing machine is defined formally as follows:

Definition 13.2 (*Turing Machine*) A Turing machine $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$ is a 7-tuple is defined as follows in [HoU:79]:

- Q is a finite set of *states*.
- Γ is a finite set of the *tape alphabet/symbols*.

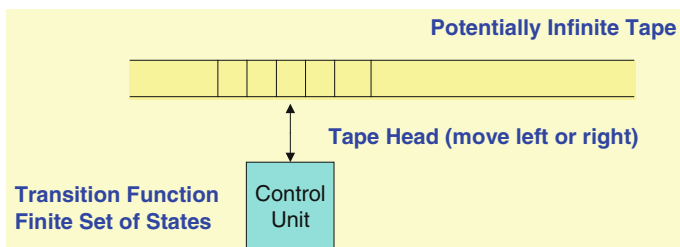


Fig. 13.7 Turing machine

- $b \in \Gamma$ is the *blank symbol* (This is the only symbol that is allowed to occur infinitely often on the tape during each step of the computation).
- Σ is the set of input symbols and is a subset of Γ (i.e. $\Gamma = \Sigma \cup \{b\}$).
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ ⁵ is the transition function. This is a partial function where L is left shift and R is right shift.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final or accepting states.

The Turing machine is a simple machine that is equivalent to an actual physical computer in the sense that it can compute exactly the same set of functions. It is much easier to analyse and prove things about than a real computer, but it is not suitable for programming and therefore does not provide a good basis for studying programming and programming languages.

Figure 13.8 illustrates the behaviour when the machine is in state q_1 and the symbol under the tape head is a , where b is written to the tape and the tape head moves to the left and the state changes to q_2 .

A Turing machine is essentially a finite-state machine (FSM) with an unbounded tape. The tape is potentially infinite and unbounded, whereas real computers have a large but finite store. The machine may read from and write to the tape. The FSM is essentially the control unit of the machine, and the tape is essentially the store. However, the storage in a real computer may be extended with backing tapes and disks and in a sense may be regarded as unbounded. However, the maximum amount of tape that may be read or written within n steps is n .

A Turing machine has an associated set of rules that defines its behaviour. Its actions are defined by the transition function. It may be programmed to solve any problem for which there is an algorithm. However, if the problem is unsolvable, then the machine will either stop or compute forever. The solvability of a problem may not be determined beforehand. There is, of course, some answer (i.e. either the machine halts or it computes forever). The applications of the Turing machine to computability and decidability are discussed in Chap. 13 of [ORg:16b].

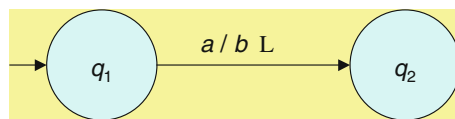


Fig. 13.8 Transition on Turing machine

⁵We may also allow no movement of the tape head to be represented by adding the symbol “N” to the set.

Turing also introduced the concept of a Universal Turing Machine, and this machine is able to simulate any other Turing machine. For more detailed information on automata theory, see [HoU:79].

13.5 Review Questions

1. What is a finite-state machine?
2. Explain the difference between a deterministic and non-deterministic finite-state machine.
3. Show how to convert the non-deterministic finite-state automaton in Fig. 7.3 to a deterministic automaton.
4. What is a pushdown automaton?
5. What is a Turing machine?
6. Explain what is meant by the language accepted by an automaton.
7. Give an example of a language accepted by a pushdown automaton but not by a finite-state machine.
8. Describe the applications of the Turing machine to computability and decidability.

13.6 Summary

Automata theory is concerned with the study of abstract machines and automata. These include finite-state machines, pushdown automata and Turing machines. Finite-state machines are abstract machines that may be in one of a finite number of states. These machines are in only one state at a time (current state), and the state transition function determines the new state from the current state and the input symbol. Finite-state machines have limited computational power due to memory and state constraints, but they have been applied to a number of fields including communication protocols and linguistics.

Pushdown automata have greater computational power than finite-state machines, and they contain extra memory in the form of a stack from which symbols may be pushed or popped. The state transition is determined from the current state of the machine, the input symbol and the element on the top of the stack. The action may be to change the state and/or push/pop an element from the stack.

The Turing machine is the most powerful model for computation, and it is equivalent to an actual computer in the sense that it can compute exactly the same set of functions. The Turing machine provides a mathematical abstraction of computer execution and storage, as well as providing a mathematical definition of an algorithm.

14.1 Introduction

Model checking is an automated technique such that given a finite state model of a system and a formal property (expressed in temporal logic), then it systematically checks whether the property is true or false in a given state in the model. It is an effective technique to identify potential design errors, and it increases the confidence in the correctness of the system design. Model checking is a highly effective verification technology and is widely used in the hardware and software fields. It has been employed in the verification of microprocessors; in security protocols; in the transportation sector (trains); and in the verification of software in the space sector.

Early work on model checking commenced in the early 1980s (especially in checking the presence of properties such as mutual exclusion and the absence of deadlocks), and the term “model checking” was coined by Clarke and Emerson [CM:81] who combined the state exploration approach and temporal logic in an efficient manner. Clarke and Emerson received the ACM Turing Award in 2007 for their role in developing model checking into a highly effective verification technology.

Model checking is a formal verification technique based on graph algorithms and formal logic. It allows the desired behaviour (specification) of a system to be verified, and its approach is to employ a suitable model of the system, and to carry out a systematic and exhaustive inspection of all states of the model to verify that the desired properties are satisfied. These properties are generally safety properties such as the absence of deadlock, request–response properties and invariants. The systematic search shows whether a given system model truly satisfies a particular property or not.

The phases in the model checking process include the modelling, running and analysis phases (Table 14.1).

Table 14.1 Model checking process

Phase	Description
Modelling phase	Model the system under consideration Formalize the property to be checked
Running phase	Run the model checker to determine the validity of the property in the model
Analysis phase	Is the property satisfied? If applicable, check next property If the property is violated then <ol style="list-style-type: none"> 1. Analyse generated counter-example 2. Refine model, design or property If out of space try alternative approach (e.g. abstraction of system model)

The model-based techniques use mathematical models to describe the required system behaviour in precise mathematical language, and the system models have associated algorithms that allow all states of the model to be systematically explored. Model checking is used for formally verifying finite state concurrent systems (typically modelled by automata), where the specification of the system is expressed in temporal logic, and efficient algorithms are used to traverse the model defined by the system (in its entirety) to check whether the specification holds or not. *Of course, any verification using model-based techniques is only as good as the underlying model of the system.*

Model checking is an automated technique such that given a finite state model of a system and a formal property, then a systematic search may be conducted to determine whether the property holds for a given state in the model. The set of all possible states is called the model's state space, and when a system has a finite state space, it is then feasible to apply model checking algorithms to automate the demonstration of properties, with a counter-example exhibited if the property is not valid.

The properties to be validated are generally obtained from the system specification, and they may be quite elementary; for example, a deadlock scenario should never arise (i.e. the system should never be able to reach a situation where no further progress is possible). The formal specification describes what the system should do, whereas the model description (often automatically generated) is an accurate and unambiguous description of how the system actually behaves. The model is often expressed in a finite state machine consisting of a finite set of states and a finite set of transitions.

Figure 14.1 shows the structure of a typical model checking system where a preprocessor extracts a state transition graph from a program or circuit. The model checking engine then takes the state transition graph and a temporal formula P and determines whether the formula is true or not in the model.

The properties need to be expressed precisely and unambiguously (usually in temporal logic) to enable rigorous verification to take place. Model checking extracts a finite model from a system and then checks some property of that model. The model checker performs an exhaustive state search, which involves checking all system states to determine whether they satisfy the desired property or not (Fig. 14.2).

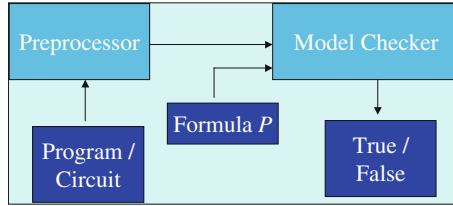
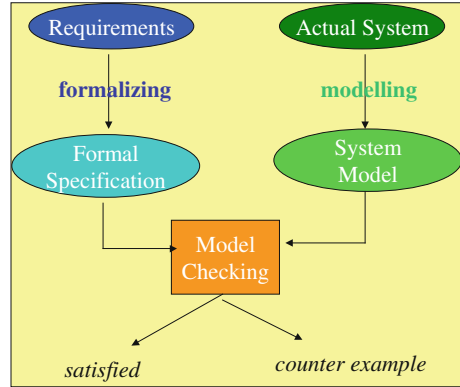


Fig. 14.1 Concept of model checking

Fig. 14.2 Model checking



If a state that violates the desired property is determined (i.e. a defect has been found once it is shown that the system does not fulfil one of its specified properties), then the model checker provides a counter-example indicating how the model can reach this undesired state. The system is considered to be correct if it satisfies all of the specified properties. In the cases of where the model is too large to fit within the physical memory of the computer (state explosion problem), then other approaches such as abstraction of the system model or probabilistic verification may be employed.

There may be several causes of a state violating the desired property. It may be due to a modelling error (i.e. the model does not reflect the design of the system, and the model may need to be corrected and the model checking restarted). Alternatively, it may be due to a design error with improvements needed to the design, or it may be due to an error in the statement of the property with a modification to the property required and the model checking needs to be restarted.

Model checking is expressed formally by showing that a desired property P (expressed as a temporal logic formula) and a model M with initial state s , that P is always true in any state derivable from s (i.e. $M, s \models P$). We discussed temporal logic briefly in Chap. 7, and model checking is concerned with verifying that linear time properties such as *safety*, *liveness* and *fairness* properties are always satisfied, and it employs *linear temporal logic* and *branching temporal logic*. *Computational tree logic* is a branching temporal logic where the model of time

is a tree-like structure, with many different paths in the future, one of which might be an actual path which is realized.

One problem with model checking is the state space explosion problem, where the transition graph grows exponentially on the size of the system, which makes the exploration of the state space difficult or impractical. Abstraction is one technique that aims to deal with the state explosion problem, and it involves creating a simplified version of the model (the abstract model). The abstract model may be explored in a reasonable period of time, and the abstract model must respect the original model with respect to key properties such that if the property is valid in the abstract model it is valid in the original model.

Model checking has been applied to areas such as the verification of hardware designs, embedded systems, protocol verification and software engineering. Its algorithms have improved over the years, and today model checking is a mature technology for verification and debugging with many successful industrial applications.

The advantages of model theory include the fact that the user of the model checker does not need to construct a correctness proof (as in automated Theorem Proving or proof checking). Essentially, all the user needs to do is to input a description of the program or circuit to be verified and the specification to be checked, and to then press the return key. The checking process is then automatic and fast, and it provides a counter-example if the specification is not satisfied. One weakness of model checking is that it verifies an actual model rather than the actual system, and so its results are only as good as the underlying model. Model checking is described in detail in [BaK:08].

14.2 Modelling Concurrent Systems

Concurrency is a form of computing in which multiple computations (processes) are executed during the same period of time. *Parallel computing* allows execution to occur in the same time instant (on separate processors of a multiprocessor machine), whereas *concurrent computing* consists of process lifetimes overlapping and where execution need not happen at the same time instant.

Concurrency employs *interleaving* where the execution steps of each process employ time-sharing slices so that only one process runs at a time, and if it does not complete within its time slice it is paused; another process begins or resumes; and then later the original process is resumed. In other words, only one process is running at a given time instant, whereas multiple processes are part of the way through execution.

It is important to identify concurrency-specific errors such as deadlock and livelock. A *deadlock* is a situation in which the system has reached a state in which no further progress can be made, and at least one process needs to complete its tasks. *Livelock* refers to a situation where the processes in a system are stuck in a repetitive task and are making no progress towards their functional goals.

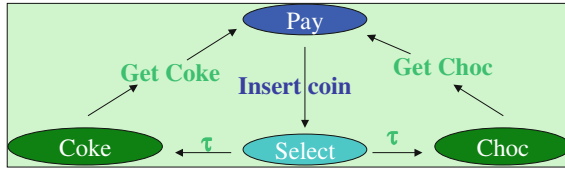


Fig. 14.3 Simple transition system

It is essential that safety properties such as *mutual exclusion* (at most one process is in its critical section at any given time) are not violated. In other words, something bad (e.g. a deadlock situation) should never happen; *liveness properties* (a desired event or something good eventually happens) are satisfied; and *invariants* (properties that are true all the time) are never violated. These behaviour errors may be mechanically detected if the systems are properly modelled and analysed.

Transition systems (Fig. 14.3) are often used as models to describe the behaviour of systems, and these are directed graphs with nodes representing *states* and edges that represent *state transitions*. A state describes information about a system at a certain moment of time. For example, the state of a sequential computer consists of the values of all program variables and the current value of the program counter (pointer to next program instruction to be executed).

A transition describes the conditions under which a system moves from one state to another. Transition systems are expressive in that programs are transition systems; communicating processes are transition systems; and hardware circuits are transition systems,

The transitions are associated with action labels that indicate the actions that cause the transition. For example, in Fig. 14.3, the *Insert coin* is a user action, whereas the *Get coke* and *Get choc* are actions that are performed by the machine. The activity τ represents an internal activity of the vending machine that is not of interest to the modeller. Formally, a transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ such that:

S	is the set of states
Act	is the set of actions
$\rightarrow S \times Act \times S$	is the transition relation (source state, action and target state)
$I \subseteq S$	is the set of initial states
AP	is a set of atomic propositions
$L: S \rightarrow \mathbb{P} AP$	is a labelling function
	(power set of AP)

The transition (s, a, s') is written as $s \xrightarrow{a} s'$

$L(s)$ are the atomic propositions in AP that are satisfied in state s .

A concurrent system consists of multiple processes executing concurrently. If a concurrent system consists of n processes where each process $proc_i$ is modelled by a transition system TS_i , then the concurrent system may be modelled by a transition system (\parallel is the parallel composition operator):

$$TS = TS_1 \parallel TS_2 \parallel \dots \parallel TS_n$$

There are various operators used in modelling concurrency with transition systems, including operators for interleaving, communication via shared variables, handshaking and channel systems.

14.3 Linear Temporal Logic

Temporal logic was discussed in Chap. 7 and is concerned with the expression of properties that have time dependencies. The existing temporal logics allow facts about the past, present and future to be expressed. Temporal logic has been applied to specify temporal properties of natural language, as well as the specification and verification of program and system behaviour. It provides a language to encode temporal knowledge in Artificial Intelligence applications, and it plays a useful role in the formal specification and verification of temporal properties (e.g. *liveness* and *fairness*) in safety critical systems.

The statements made in temporal logic can have a truth-value that varies over time. In other words, sometimes the statement is true and sometimes it is false, but it is never true or false at the same time. The two main types of temporal logics are *linear time logics* (reason about a single time line) and *branching time logics* (reason about multiple timelines).

Linear temporal logic (LTL) is a modal temporal logic that can encode formulae about the future of paths (e.g. a condition that will eventually be true). The basic linear temporal operators that are often employed (p is an atomic proposition below) are listed in Table 14.2 and illustrated in Fig. 14.4.

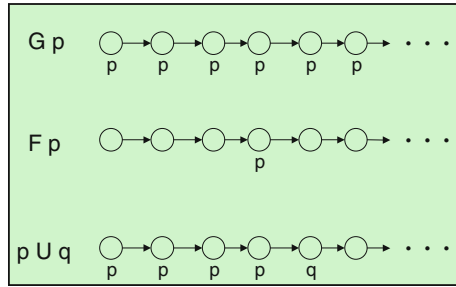
For example, consider how the sentence “*This microwave does not heat up until the door is closed*” is expressed in temporal logic. This is naturally expressed with the until operator $p \cup q$ as follows:

$$\neg \text{Heatup} \cup \text{DoorClosed}$$

Table 14.2 Basic temporal operators

Operator	Description
F_p	p holds sometime in the future
G_p	p holds globally in the future
X_p	p holds in next time instant
$p \cup q$	p holds until q is true

Fig. 14.4 LTL operators



14.4 Computational Tree Logic

In linear logic, we look at the execution paths individually, whereas in branching time logics, we view the computation as a tree. Computational tree logic (CTL) is a branching time logic, which means that its model of time is a tree-like structure in which the future is not determined, and so there are many paths in the future such that any of them could be an actual path that is realized. CTL was first proposed by Clark and Emerson in the early 1980s.

Computational tree logic can express many properties of finite state concurrent systems. Each operator of the logic has two parts, namely the path quantifier (A —“every path”, E—“there exists a path”) and the state quantifier (F, P, X, U as explained in Table 14.3). The operators in CTL logic are given by:

For example, the following is a valid CTL formula that states that it is always possible to get to the restart state from any state:

$$AG(EF restart)$$

Table 14.3 CTL temporal operators

Operator	Description
$A\varphi$ (<i>all</i>)	φ holds on <i>all</i> paths starting from the current state
$E\varphi$ (<i>exists</i>)	φ holds on at least one path starting from the current state
$X\varphi$ (<i>next</i>)	φ holds in the next state
$G\varphi$ (<i>global</i>)	φ has to hold on the entire subsequent path
$F\varphi$ (<i>finally</i>)	φ eventually has to hold (somewhere on the subsequent path)
$\varphi \cup \psi$ (<i>until</i>)	φ has to hold until at some position ψ holds
$\varphi W\psi$ (<i>weak until</i>)	φ has to hold until ψ holds (no guarantee ψ will ever hold)

14.5 Tools for Model Checking

There are various tools for model checking including Spin, Bandera, SMV and UppAal. These tools perform a systematic check on property P in all states and are applicable if the system generates a finite behavioural model. Model checking tools use a model-based approach rather than a proof-rule-based approach, and the goal is to determine whether the concurrent program satisfies a given logical property.

Spin is a popular open-source tool that is used for the verification of distributed software systems (especially concurrent protocols), and it checks finite state systems with properties specified by linear temporal logic. It generates a counter-example trace if it determines that a property is violated.

Spin has its own input specification language (PROMELA), and so the system to be verified needs to be translated to the language of the model checker. The properties are specified using LTL.

Bandera is a tool for model checking Java source code, and it automates the extraction of a finite state model from the Java source code. It then translates into an existing model checker's input language. The properties to be verified are specified in the Bandera Specification Language (BSL), which supports precondition and postcondition and temporal properties.

14.6 Industrial Applications of Model Checking

There are many applications of model checking in the hardware and software fields, including the verification of microprocessors and security protocols, as well as applications in the transportation sector (trains) and in the space sector.

The Mars Science Laboratory (MSL) mission used model checking as part of the verification of the critical software for the landing of Curiosity (a large rover) in its mission to Mars. The hardware and software of a spacecraft must be designed for a high degree of reliability, as an error can lead to a loss of the mission. The Spin model checker was employed for the model verification, and the rover landed safely on 5 August 2012.

CMG employed formal methods as part of the specification and verification of the software for a movable flood barrier, which is used to protect the port of Rotterdam from flooding. Z was employed for modelling data, and operations and Spin/Promela were used for model checking.

Lucent's Pathstar Access Server was developed in the late 1990s, and this system is capable of sending voice and data over the Internet. Formal methods are often employed in the telecommunications sector in dealing with the feature interaction problem. The automated verification techniques applied to Pathstar

consist of generating an abstract model from the implemented C code, and then defining the formal requirements that the application is satisfy. Finally, the model checker is employed to perform the verification.

14.7 Review Questions

1. What is model checking?
2. Explain the state explosion problem.
3. Explain the difference between parallel processing and concurrency.
4. Describe the basic temporal operators.
5. Describe the temporal operators in CTL.
6. Explain the difference between liveness and fairness properties.
7. What is a transition system?
8. Explain the difference between linear temporal logic and branching temporal logic.
9. Investigate tools to support model checking.

14.8 Summary

Model checking is a formal verification technique which allows the desired behaviours of a system to be verified. Its approach is to employ a suitable model of the system and to carry out a systematic inspection of all states of the model to verify the properties. The properties to be validated are generally obtained from the system specification, a defect is found once it is shown that the system does not fulfil one of its specified properties, and the system is considered to be correct if it satisfies all of the specified properties.

It allows the desired behaviour (specification) of a system to be verified, and its approach is to employ a suitable model of the system and to carry out a systematic and exhaustive inspection of all states of the model to verify that the desired properties are satisfied. These properties are generally properties such as the absence of deadlock and invariants. The systematic search shows whether a given system model truly satisfies a particular property or not.

The model-based techniques use mathematical models to describe the required system behaviour in precise mathematical language, and the system models have associated algorithms that allow all states of the model to be systematically

explored. The specification of the system is expressed in temporal logic, and efficient algorithms are used to traverse the model defined by the system (in its entirety) to check whether the specification holds or not. Model-based techniques are only as good as the underlying model of the system.

15.1 Introduction

The word *proof* is generally interpreted as arguments in natural language that presents facts or evidence to establish a particular conclusion. Several premises (which are already established) are presented, and from these premises (via deductive or inductive reasoning), further propositions are established, until finally the conclusion is established.

A *mathematical proof* typically includes natural language and mathematical symbols, and often, many of the tedious details of the proof are omitted. Mathematical proof dates back to the Greeks, and most students are familiar with Euclid's work (*The Elements*) in geometry, where from a small set of axioms and postulates and definitions he derived many of the well-known theorems of geometry. Euclid was a Hellenistic mathematician based in Alexandria around 300BC, and his style of proof was mainly constructive; that is, in addition to the proof of the existence of an object, he actually constructed the object in the proof. Euclidean geometry remained unchallenged for over 2000 years, until the development of the non-Euclidean geometries in the nineteenth century, and these geometries were based on a rejection of Euclid's controversial 5th postulate (the parallels postulate).

Mathematical proof may employ a "*divide and conquer*" technique, i.e., breaking the conjecture down into subgoals and then attempting to prove each of the subgoals. Another common proof technique is *indirect proof* where we assume the opposite of what we wish to prove, and we show that this results in a contradiction (e.g. consider the proof that there are an infinite number of primes or the proof that there is no rational number whose square is 2). Other proof techniques used are the method of mathematical induction, where involves a proof of the base case and inductive step.

Aristotle developed *sylogistic logic* in the fourth-century BC, and the rules of reasoning with valid syllogisms remained dominant in logic up to the nineteenth century. Boole develops his mathematical logic in the mid-nineteenth century, and he aimed to develop a calculus of reasoning to verify the correctness of arguments

using logical connectives. Predicate logic (including universal and existential quantifiers) was introduced by Frege in the late nineteenth century as part of his efforts to derive mathematics from purely logical principles. Russell and Whitehead continued this attempt in *Principia Mathematica*, and Russell introduced the theory of types to deal with the paradoxes in set theory, which he identified in Frege's system.

The *formalists* introduced extensive axioms in addition to logical principles, and Hilbert's program led to the definition of a *formal mathematical proof* as a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules (e.g. *modus ponens*). The last line in the proof is the theorem to be proved, and the formal proof is essentially syntactic following rules with the formulae simply a string of symbols and the meaning of the symbols is unimportant.

The formalists later ran into problems in trying to prove that a formal system powerful enough to include arithmetic was both complete and consistent, and the results of Gödel showed that such a system would be *incomplete* (and one of the propositions without a proof is that of its own *consistency*). Turing later showed (with his Turing machine) that mathematics is *undecidable*; that is, there is no algorithm or mechanical procedure that may be applied in a finite number of steps to determine whether an arbitrary mathematical proposition is true or false.

The proofs employed in mathematics are rarely formal (in the sense of Hilbert's program), and whereas they involve deductions from a set of axioms, these deductions are rarely expressed as the application of individual rules of logical inference. Many proofs in formal methods are concerned with cross-checking the details of the specification, or checking the validity of the refinement steps, or checking that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers¹ are essential in dealing with these. Machine proof is explicit, and reliance on some brilliant insight is avoided. Proofs by hand in formal methods are notorious for containing errors or jumps in reasoning, whereas machine proofs are explicit but are often extremely lengthy and essentially unreadable.

The application of formal methods in an industrial environment requires the use of machine-assisted proof, since thousands of proof obligations arise from a formal specification, and theorem provers are essential in resolving these efficiently. Automated theorem proving (ATP) is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

¹Most existing theorem provers are difficult to use and are for specialist use only. There is a need to improve the usability of theorem provers.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness² is unlikely except for the most trivial of programs. A program may consist of legacy software that is assumed to work; a compiler that is assumed to work correctly creates it. Theorem provers are programs that are assumed to function correctly. The best that formal methods can claim is increased confidence in correctness of the software, rather than an absolute proof of correctness.

15.2 Early Automation of Proof

Early work on the automation of proof began in the 1950s with the beginning of work in the Artificial Intelligence field, where the early AI practitioners were trying to develop a “*thinking machine*”. One of the earliest programs developed was the *Logic Theorist*, which was presented at the Dartmouth conference on Artificial Intelligence in 1956 [MacK:95].

It was developed by Allen Newell and Herbert Simon, and it could prove 38 of the first 52 theorems from Russell and Whitehead’s *Principia Mathematica*.³ The Logic Theorist proved theorems in the propositional calculus (see Chap.6), but did not support predicate calculus. It used the five basic axioms of propositional logic and three rules of inference from the *Principia* to prove theorems. It was programmed to start with the theorem to be proved and to then search for a proof of it (Fig. 15.1).⁴

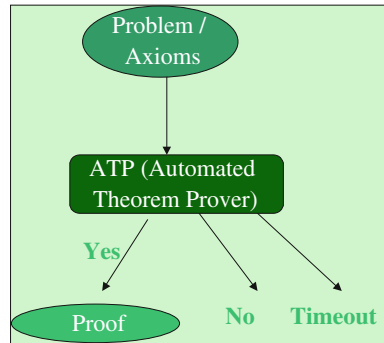
If no immediate one-step proof could be found, then a set of subgoals was generated (these are formulae from which the theorem may be proved in one step) and proofs of these were then searched for, and so on. The program could use previously proved theorems in the course of developing a proof of a new theorem. Newell and Simon were hoping that the Logic Theorist would do more than just prove theorems in logic, and their goal was that it would attempt to prove theorems in a human-like way and especially with a selective search.

²This position is controversial with others arguing that if correctness is defined mathematically, then the mathematical definition (i.e. formal specification) is a theorem, and the task is to prove that the program satisfies the theorem. They argue that the proofs for non-trivial programs exist and that the reason why there are not many examples of such proofs is due to a lack of mathematical specifications.

³Russell is said to have remarked that he was delighted to see that the Principia Mathematica could be done by machine and that if he and Whitehead had known this in advance that they would not have wasted 10 years doing this work by hand in the early twentieth century.

⁴Another possibility (though an inefficient and poor simulation of human intelligence) would be to start with the five axioms of the *Principia* and to apply the three rules of inference to logically derive all possible sequences of valid deductions. This is known as the British Museum algorithm (as sensible as putting monkeys in front of typewriters to reproduce all of the books of the British Museum).

Fig. 15.1 Idea of automated theorem proving



However, in practice, the Logic Theorist search was not very selective in its approach, and the subproblems were considered in the order in which they were generated, and so there was no actual heuristic procedure (as in human problem-solving) to guess at which subproblem was most likely to yield an actual proof. This meant that the Logic Theorist could, in practice, find only very short proofs, since as the number of steps in the proof increased, the amount of search required to find the proof exploded.

The Geometry Machine was developed by Herbert Gelernter at the IBM Research Center in New York in the late 1950s, with the goal of developing intelligent behaviour in machines. It differed from the Logic Theorist in that it selected only the valid subgoals (i.e. it ignored the invalid ones) and attempted to find a proof of these. The Geometry Machine was successful in finding the solution to a large number of geometry problems taken from high-school textbooks in plane geometry.

The logicians Hao Wang and Evert Beth (the inventor of semantic tableaux which was discussed in Chap. 6) were critical of the approaches of the AI pioneers and believed that mathematical logic could do a lot more. Wang and others developed a theorem prover for first-order predicate calculus in 1960, but it had serious limitations due to the combinatorial explosion.

Alan Robinson's work on theorem provers in the early 1960s led to a proof procedure termed "*resolution*", which appeared to provide a breakthrough in the automation of predicate calculus theorem provers. A resolution theorem prover is essentially provided with the axioms of the field of mathematics in question and the negation of the conjecture whose proof is sought. It then proceeds until a contradiction is reached, where there is no possible way for the axioms to be true and for the conjecture to be false.

The initial success of resolution led to excitement in the AI field where pioneers such as John McCarthy (see Chap. 7) believed that human knowledge could be expressed in predicate calculus⁵ and that therefore if resolution was indeed successful for efficient automated theorem provers, then the general problem of Artificial Intelligence was well on the way to a solution. However, while resolution led to improvements with the state explosion problem, it did not eliminate the problem.

This led to a fall off in research into resolution-based approaches to theorem proving, and other heuristic-based techniques were investigated by Bledsoe in the late 1970s. The field of logic programming began in the early 1970s with the development of the Prolog programming language. Prolog is in a sense an application of automated theorem proving, where problems are stated in the form of goals (or theorems) in which the system tries to prove using a resolution theorem prover. The theorem prover generally does not need to be very powerful as many Prolog programs require only a very limited search, and a *depth-first* search from the goal backwards to the hypotheses is conducted.

The Argonne Laboratory developed the Aura System in the early 1980s (it was later replaced by Otter), as an improved resolution-based automated theorem prover, and this led to renewed interest in resolution-based approaches to theorem proving. There is a more detailed account of the nature of proof and theorem proving in [MacK:95].

15.3 Interactive Theorem Provers

The challenges in developing efficient automated theorem provers led researchers to question whether an effective fully automated theorem prover was possible, and whether it made more sense to develop a theorem prover that could be guided by a human in its search for a proof. This led to the concept of *interactive theorem proving* (ITP) which involves developing formal proofs by man-machine collaboration and is (in a sense) a new way of doing mathematics in front of a computer.

Such a system is potentially useful in mathematical research in formalizing and checking proofs, and it allows the user to concentrate on the creative parts of the proof and relieves the user of the need of carrying out the trivial steps in the proof. It is also a useful way of verifying the correctness of published mathematical proofs by acting as a *proof checker*, where the ITP is provided with a formal proof constructed by a human, which may then be checked for correctness.⁶ Such a

⁵McCarthy's viewpoint that predicate logic was the solution for the AI field was disputed by Minsky and others (resulting in a civil war between the logicians and the proceduralists). The proceduralists argued that formal logic was an inadequate representation of knowledge for AI and that predicate calculus was an overly rigid and inadequate framework. They argued that an alternative approach such as the procedural representation of knowledge was required.

⁶A formal mathematical proof (of a normal proof) is difficult to write down and can be lengthy. Mathematicians were not really interested in these proof checkers.

system is important in *program verification* in showing that the program satisfies its specification, especially in the safety/security critical fields.

A group at Princeton developed a series of systems called semi-automated mathematics (SAM) in the late 1960s, which combined logic routines with human guidance and control. Their approach placed the mathematician at the heart of the theorem proving, and it was a departure from the existing theorem proving approaches where the computer attempted to find proofs unaided. SAM provided a proof of an unproven conjecture in lattice theory (SAM's lemma), and this is regarded as the first contribution of automated reasoning systems to mathematics [MacK:95].

De Bruijn and others at the Technics Hogeschool in Eindhoven in the Netherlands commenced development of the Automath system in the late 1960s. This was a large-scale project for the automated verification of mathematics, and it was tested by treating a full textbook. Automath systematically checked the proofs from Landau's text *Grundlagen der Analysis* (this foundation of analysis text was first published in 1930).

The typical components of an interactive theorem prover include an interactive proof editor to allow editing of proofs, formulae and terms in a formal theory of mathematics, and a large library of results which is essential for achieving complex results.

The Gypsy verification environment and its associated theorem prover was developed at the University of Texas in the 1980s, and it achieved early success in program verification with its verification of the encrypted packet interface program (a 4200 line program). It supports the development of software systems and formal mathematical proof of their behaviour.

The Boyer-Moore theorem prover (NQTHM) was developed in the 1970/1980s at the University of Texas. It supports mathematical induction as a rule of inference, and induction is a useful technique in proving properties of programs. The axioms of Peano arithmetic are built into the theorem prover, and new axioms added to the system need to pass a "*correctness test*" to prevent the introduction of inconsistencies. It is far more automated than many other interactive theorem provers, but it requires detailed human guidance (with suggested lemmas) for difficult proofs. The user therefore needs to understand the proof being sought and the internals of the theorem prover. Many mathematical theorems have been proved including Gödel's incompleteness theorem.

The HOL system was developed at the University of Cambridge, and it is an environment for interactive theorem proving in a higher-order logic. It requires skilled human guidance and has been used for the verification of microprocessor design. It is one of the most widely used theorem provers.

15.4 A Selection of Theorem Provers

Table 15.1 presents a small selection of the available automated and interactive theorem provers.

Table 15.1 Selection of theorem provers

Theorem prover	Description
ACL2	A Computational Logic for Applicative Common Lisp (ACL2) is part of the Boyer-Moore family of theorem provers. It is a software system consisting of a programming language (LISP) and an interactive theorem prover. It was developed in the mid-1990s as an industrial strength successor to the Boyer-Moore theorem prover (NQTHM). It is used in the verification of safety-critical hardware and software and in industrial applications such as the verification of the floating point module of a microprocessor
OTTER	OTTER is a resolution-style theorem prover for first-order logic developed at the Argonne Laboratory at the University of Chicago (it was the successor to Aura). It has been mainly applied to abstract algebra and formal logic
PVS	The Prototype Verification System (PVS) is a mechanized environment for formal specification and verification. It includes a specification language integrated with support tools and an interactive theorem prover. It was developed by SRI in California. The specification language is based on higher-order logic, and the theorem prover is guided by the user in conducting proof. It has been applied to the verification of hardware and software
Theorem Proving System (TPS)	TPS is an automated theorem prover for first-order logic and higher-order logic (it can also prove theorems interactively). It was developed at Carnegie Mellon University and is used for hardware and software verification
HOL and Isabelle	HOL and Isabelle were developed by the Automated Reasoning Group at the University of Cambridge. The HOL system is an environment for interactive theorem proving in a higher-order logic, and it has been applied to hardware verification. Isabelle is a generic proof assistant which allows mathematical formulae to be expressed in a formal language, and it provides tools for proving those formulae in a logical calculus
Boyer-Moore	The Boyer-Moore theorem prover (NQTHM) was developed at the University of Texas in the 1970s with the goal of checking the correctness of computer systems. It has been used to verify the correctness of microprocessors, and it has been superseded by ACL2

15.5 Review Questions

1. What is a mathematical proof ?
2. What is a formal mathematical proof ?
3. What approaches are used to prove a theorem?
4. What is a theorem prover?
5. What role can theorem provers play in software development?
6. What is the difference between an automated theorem prover and an interactive theorem prover?
7. Investigate and give a detailed description of one of the theorem provers in Table 15.1.

15.6 Summary

A mathematical proof typically includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. The proofs employed in mathematics are rarely formal as such, and many proofs in formal methods are concerned with cross-checking the details of the specification, or checking the validity of the refinement steps, or checking that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers are essential in dealing with these. Machine proof is explicit, and reliance on some brilliant insight is avoided. Proofs by hand often contain errors or jumps in reasoning, while machine proofs are often extremely lengthy and unreadable.

The application of formal methods in an industrial environment requires the use of machine-assisted proof, since thousands of proof obligations arise from a formal specification, and theorem provers are essential in resolving these efficiently. Automated theorem proving is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness is unlikely except for the most trivial of programs. The best that formal methods can claim is increased confidence in correctness of the software, rather than an absolute proof of correctness.

Early work on the automation of proof began in the 1950s with the beginning of work in the Artificial Intelligence field, and one of the earliest programs developed was the Logic Theorist, which was presented at the Dartmouth conference on Artificial Intelligence in 1956.

The challenges in developing effective automated theorem provers led researchers to investigate whether it made more sense to develop a theorem prover that could be guided by a human in its search for a proof. This led to the development of interactive theorem proving which involved developing formal proofs by man-machine collaboration.

The typical components of an interactive theorem prover include an interactive proof editor to allow editing of proofs, formulae and terms in a formal theory of mathematics, and a large library of results which is essential for achieving complex results.

An interactive theorem prover allows the user to concentrate on the creative parts of the proof and relieves the user of the need to carry out and verify the trivial steps in the proof. It is also a useful way of verifying the correctness of published mathematical proofs by acting as a proof checker and is also useful in program verification in showing that the program satisfies its specification, especially in the safety/security critical fields.

16.1 Introduction

Statistics is an empirical science that is concerned with the collection, organization, analysis, interpretation and presentation of data. The data collection needs to be planned, and this may include surveys and experiments. Statistics is widely used by government and industrial organizations, and they may be employed for forecasting as well as for presenting trends. They allow the behaviour of a population to be studied and inferences to be made about the population. These inferences may be tested (*hypothesis testing*) to ensure their validity.

The analysis of statistical data allows an organization to understand its performance in key areas, and to identify problematic areas. Organizations will often examine performance trends over time, and will devise appropriate plans and actions to address problematic areas. The effectiveness of the actions taken will be judged by improvements in performance trends over time.

It is often not possible to study the entire population, and instead a representative subset or sample of the population is chosen. This *random sample* is used to make inferences regarding the entire population, and it is essential that the sample chosen

is indeed random and representative of the entire population. Otherwise, the inferences made regarding the entire population will be invalid.¹

A statistical experiment is a causality study that aims to draw a conclusion regarding values of a *predictor variable(s)* on a *response variable(s)*. For example, a statistical experiment in the medical field may be conducted to determine if there is a causal relationship between the use of a particular drug and the treatment of a medical condition such as lowering of cholesterol in the population. A statistical experiment involves:

- Planning the research;
- Designing the experiment;
- Performing the experiment;
- Analysing the results;
- Presenting the results.

Probability is a way of expressing the likelihood of a particular event occurring. It is normal to distinguish between the frequency interpretation and the subjective interpretation of probability [Ros:87]. For example, if a geologist states that “there is a 70% chance of finding gas in a certain region”, then this statement is usually interpreted in two ways:

- The geologist is of the view that over the long run 70% of the regions whose environment conditions are very similar to the region under consideration have gas [*frequency interpretation*].
- The geologist is of the view that it is likely that the region contains gas, and that 0.7 is a measure of the geologist’s belief in this hypothesis [*personal interpretation*].

However, the mathematics of probability is the same for both the frequency and personal interpretation.

16.2 Probability Theory

Probability theory provides a mathematical indication of the likelihood of an event occurring, and the probability of an event is a numerical value between 0 and 1. A probability of 0 indicates that the event cannot occur, whereas a probability of 1 indicates that the event is guaranteed to occur. If the probability of an event is greater than 0.5, then this indicates that the event is more likely to occur than not to occur.

¹The random sample leads to predictions for the entire population that may be inaccurate. For example, consider the opinion polls on the 2016 British Referendum on EU membership and the 2016 Presidential Election in the USA.

A *sample space* is the set of all possible outcomes of an experiment, and an *event* E is a subset of the sample space. For example, the sample space for the experiment of tossing a coin is the set of all possible outcomes of this experiment, i.e. head or tails. The event that the toss results a tail is a subset of the sample space.

$$S = \{h, t\} \quad E = \{t\}.$$

Similarly, the sample space for the gender of a newborn baby is the set of outcomes; i.e. the newborn baby is a boy or a girl. The event that the baby is a girl is a subset of the sample space.

$$S = \{b, g\} \quad E = \{g\}.$$

For any two events E and F of a sample space S , we can also consider the union and intersection of these events. That is,

- $E \cup F$ consists of all outcomes that are in E or F or both.
- $E \cap F$ (normally written as EF) consists of all outcomes that are in both E and F .
- E^c denotes the complement of E with respect to S and represents the outcomes of S that are not in E (i.e. $S \setminus E$).

If $EF = \emptyset$, then there are no outcomes in both E and F , and so the two events E and F are mutually exclusive. The union and intersection of two events can be extended to the union and intersection of a family of events E_1, E_2, \dots, E_n (i.e. $\cup_{i=1}^n E_i$ and $\cap_{i=1}^n E_i$).

16.2.1 Laws of Probability

The laws of probability essentially state that the probability of an event is between 0 and 1, and that the probability of the union of a mutually disjoint set of events is the sum of their individual probabilities.

- (i) $P(S) = 1$
- (ii) $P(\emptyset) = 0$
- (iii) $0 \leq P(E) \leq 1$
- (iv) For any sequence of mutually exclusive events E_1, E_2, \dots, E_n . (i.e. $E_i E_j = \emptyset$, where $i \neq j$), then the probability of the union of these events is the sum of their individual probabilities; i.e.

$$P\left(\cup_{i=1}^n E_i\right) = \sum_{i=1}^n P(E_i).$$

The probability of the union of two events (not necessarily disjoint) is given by:

$$P(E \cup F) = P(E) + P(F) - P(EF).$$

The probability of an event E not occurring is denoted by E^c and is given by $1 - P(E)$. The probability of an event E occurring given that an event F has occurred is termed the *conditional probability* (denoted by $P(E|F)$) and is given by:

$$P(E|F) = \frac{P(EF)}{P(F)} \quad \text{where } P(F) > 0.$$

This formula allows us to deduce that:

$$P(EF) = P(E|F)P(F).$$

Bayes' formula enables the probability of an event E to be determined by a weighted average of the conditional probability of E given that the event F occurred and the conditional probability of E given that F has not occurred:

$$\begin{aligned} E &= E \cap S = E \cap (F \cup F^c) \\ &= EF \cup EF^c \end{aligned}$$

$$\begin{aligned} P(E) &= P(EF) + P(EF^c) \quad (\text{since } EF \cap EF^c = \emptyset) \\ &= P(E|F)P(F) + P(E|F^c)P(F^c) \\ &= P(E|F)P(F) + P(E|F^c)(1 - P(F)). \end{aligned}$$

Two events E, F are *independent* if knowledge that F has occurred does not change the probability that E has occurred. That is, $P(E|F) = P(E)$, and since $P(E|F) = P(EF)/P(F)$, we have that two events E, F are independent if:

$$P(EF) = P(E)P(F).$$

Two events E and F that are not independent are said to be *dependent*.

16.2.2 Random Variables

Often, some numerical quantity determined by the result of the experiment is of interest rather than the result of the experiment itself. These numerical quantities are termed *random variables*. A random variable is termed *discrete* if it can take on a finite or countable number of values; otherwise, it is termed *continuous*.

The *distribution function* of a random variable is the probability that the random variable X takes on a value less than or equal to x . It is given by:

$$F(x) = P\{X \leq x\}.$$

All probability questions about X can be answered in terms of its distribution function F . For example, the computation of $P\{a < X < b\}$ is given by:

$$F(a) = \sum_{\forall x \leq a} p(x)$$

The probability mass function for a discrete random variable X (denoted by $p(a)$) is the probability that it is a certain value. It is given by:

$$p(a) = P\{X = a\}.$$

Further, $F(a)$ can also be expressed in terms of the probability mass function

$$F(a) = \sum_{\forall x \leq a} p(x)$$

We may also define a probability density function and a probability distribution function for a continuous random variable X [ORg:12], and all probability statements about X can be answered in terms of its density function $f(x)$, and the derivative of the probability distribution function yields the probability density function.

The expected value (i.e. the *mean*) of a discrete random variable X (denoted $E[X]$) is given by the weighted average of the possible values of X , and the expected value of a function of a random variable is given by $E[g(X)]$. These are defined as:

$$E[X] = \sum_i x_i P\{X = x_i\}$$

$$E[g(X)] = \sum_i g(x_i) P\{X = x_i\}.$$

The *variance* of a random variable is a measure of the spread of values from the mean, and is defined by:

$$\text{Var}(X) = E[X^2] - (E[X])^2.$$

The standard deviation σ is given by the square root of the variance. That is,

$$\sigma = \sqrt{\text{Var}(X)}.$$

The *covariance* of two random variables is a measure of the relationship between two random variables X and Y , and indicates the extent to which they both change (in either similar or opposite ways) together. It is defined by:

$$\text{Cov}(X, Y) = E[XY] - E[X]E[Y].$$

It follows from the definition that the covariance of two independent random variables is zero (and this would be expected as a change in one variable would not affect the other). Variance is a special case of covariance (when the two random variables are identical). This follows since $\text{Cov}(X, X) = E[X \cdot X] - (E[X])^2$ ($E[X] = E[X^2] - (E[X])^2 = \text{Var}(X)$).

A positive covariance ($\text{Cov}(X, Y) \geq 0$) indicates that Y tends to increase as X does, whereas a negative covariance indicates that Y tends to decrease as X increases. The *correlation* of two random variables is an indication of the relationship between two variables X and Y . If the correlation is negative, then Y tends to decrease as X increases, and if it is positive number, then Y tends to increase as X increases. The correlation coefficient is a value that is between ± 1 , and it is defined by:

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}.$$

Once the correlation between two variables has been calculated, the probability that the observed correlation was due to chance can be computed. This is to ensure that the observed correlation is a real one and not due to a chance occurrence.

There are a number of special random variables, and these include the Bernoulli trial, where there are just two possible outcomes of an experiment, i.e. success or failure. The probability of success and failure is given by:

$$\begin{aligned} P\{X = 0\} &= 1 - p \\ P\{X = 1\} &= p \end{aligned}$$

The mean of the Bernoulli distribution is given by p and the variance by $p(1-p)$. The *binomial distribution* involves n Bernoulli trials, each of which results in success or failure. The probability of i successes from n trials is then given by:

$$P\{X = i\} = \binom{n}{i} p^i (1-p)^{n-i}.$$

where the mean of the binomial distribution is given by np , and the variance is given by $np(1-p)$.

The *Poisson distribution* may be used as an approximation to the binomial distribution when n is large and p is small. The probability of i successes is given by:

$$P\{X = i\} = e^{-\lambda} \lambda^i / i!$$

and the mean and variance of the Poisson distribution is given by λ .

There are many other well-known distributions such as the *hypergeometric distribution* that describes the probability of i successes in n draws from a finite population without replacement; the *uniform distribution*; the *exponential*

Table 16.1 Probability distributions

Distribution name	Density function	Mean/variance
Binomial	$P\{X = i\} = \binom{n}{i} p^i (1-p)^{n-i}$	$np, np(1-p)$
Poisson	$P\{X = i\} = e^{-\lambda} \lambda^i / i!$	λ, λ
Hypergeometric	$P\{X = i\} = \binom{N}{i} \binom{M}{n-i} / \binom{N+M}{n}$	$nN/N + M, np(1-p)[1 - (n-1)/N + M - 1]$
Uniform	$f(x) = 1/(\beta - \alpha) \alpha \leq x \leq \beta, 0$	$(\alpha + \beta)/2, (\beta - \alpha)^2/12$
Exponential	$f(x) = \lambda e^{-\lambda x}$	$1/\lambda, 1/\lambda^2$
Normal	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$	μ, σ^2
Gamma	$f(x) = \lambda e^{-\lambda x} (\lambda x)^{\alpha-1} \Gamma(\alpha) (x \geq 0)$	$\alpha/\lambda, \alpha/\lambda^2$

distribution; the *normal distribution* and the *gamma distribution*. Table 16.1 presents several important probability distributions including their mean and variance:

The reader is referred to [Ros:87] for a more detailed account of probability theory.

16.3 Statistics

The field of statistics is concerned with summarizing, digesting and extracting information from large quantities of data. Statistics provides a collection of methods for planning and conducting an experiment, and analysing the data to draw accurate conclusions. We distinguish between descriptive statistics and inferential statistics:

Descriptive Statistics

This is concerned with describing the information in a set of data elements in graphical format, or by describing its distribution.

Inferential Statistics

This is concerned with making inferences with respect to the population by using information gathered in the sample.

16.3.1 Abuse of Statistics

Statistics is extremely useful in drawing conclusions about a population. However, it is essential that the random sample is valid, and that the experiment is properly conducted to enable valid conclusions to be inferred. Further, the presentation of the statistics should not be misleading. Some examples of the abuse of statistics include:

- The sample size may be too small to draw conclusions.
- It may not be a genuine random sample of the population.

- Graphs may be drawn to exaggerate small differences.
- Area may be misused in representing proportions.
- Misleading percentages may be used.

The quantitative data used in statistics may be discrete or continuous. *Discrete data* is numerical data that has a finite number of possible values, and *continuous data* is numerical data that has an infinite number of possible values.

16.3.2 Statistical Sampling

Statistical sampling is concerned with the methodology of choosing a random sample of a population, and the systematic study of the sample with the goal of drawing valid conclusions about the entire population.

The assumption is that if a genuine representative sample of the population is chosen, then the detailed study of the sample will provide insight into the whole population. This helps to avoid a lengthy expensive (and potentially infeasible) study of the entire population.

The sample chosen must be random (this can be difficult to achieve), and the sample size is sufficiently large to enable valid conclusions to be made for the entire population.

Random Sample

A *random sample* is a sample of the population such that each member of the population has an equal chance of being chosen.

There are various ways of generating a random sample from the population including (Table 16.2):

Once the random sample group has been chosen, the next step is to obtain the required information from the sample. This may be done by interviewing each member in the sample; phoning each member; conducting a mail survey and so on (Table 16.3).

Table 16.2 Sampling techniques

Sampling techniques	Description
Systematic sampling	Every <i>k</i> th member of the population is sampled
Stratified sampling	The population is divided into two or more strata, and each subpopulation (stratum) is then sampled. Each element in the subpopulation shares the same characteristics (e.g. age groups, gender)
Cluster sampling	A population is divided into clusters, and a few of these clusters are exhaustively sampled (i.e. every element in the cluster is considered)
Convenience sampling	Sampling is done as convenient and often allows the element to choose whether or not it is sampled

Table 16.3 Types of survey

Survey type	Description
Direct measurement	This may involve the direct measurement of all entities in the sample (e.g. the height of students in a class)
Mail survey	This involves sending a mail survey to the sample. This may have a lower response rate and may thereby invalidate the findings
Phone survey	This is a reasonably efficient and cost-effective way to gather data. However, refusals or hang-ups may affect the outcome
Personal interview	This tends to be expensive and time-consuming, but it allows detailed information to be collected
Observational study	An observational study allows individuals to be studied, and the variables of interest to be measured
Experiment	An experiment imposes some treatment on individuals in order to study the response

16.3.3 Averages in a Sample

The term “average” generally refers to the arithmetic *mean* of a sample, but it could also refer to the *mode* or *median* of the sample. These terms are defined below:

Mean

The *arithmetic mean* of a set of n numbers is defined to be the sum of the numbers divided by n . That is, the arithmetic mean for a sample of size n is given by:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}.$$

The actual \bar{x} mean of the population is denoted by μ , and it may differ from the sample mean.

Mode

The mode is the data element that occurs most frequently in the sample. It is possible that two elements occur with the same frequency, and if this is the case, then we are dealing with a bi-modal or possibly a multi-modal sample.

Median

The median is the middle element when the data set is arranged in increasing order of magnitude.

If there are an odd number of elements in the sample, the median is the middle element. Otherwise, the median is the arithmetic mean of the two middle elements.

Mid-Range

The mid-range is the arithmetic mean of the highest and lowest data elements in the sample. That is, $(x_{\max} + x_{\min})/2$.

The *arithmetic mean* is the most widely used average in statistics.

16.3.4 Variance and Standard Deviation

An important characteristic of a sample is its distribution, and the spread of each element from some measure of central tendency (e.g. the mean). One elementary measure of dispersion is that of the sample *range*, and it is defined to be the difference between the maximum and minimum value in the sample. That is, the sample range is defined to be:

$$\text{range} = x_{\max} - x_{\min}.$$

The sample range is not a reliable measure of dispersion as only two elements in the sample are used, and extreme values in the sample can distort the range to be very large even if most of the elements are quite close to one another.

The standard deviation is the most common way to measure dispersion, and it gives the average distance of each element in the sample from the mean. The sample standard deviation is denoted by s and is defined by:

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

The population standard deviation is denoted by σ and is defined by:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Variance is another measure of dispersion, and it is defined as the square of the standard deviation. The sample variance is given by:

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n - 1}$$

The population variance is given by:

$$\sigma^2 = \frac{\sum (x_i - \mu)^2}{N}$$

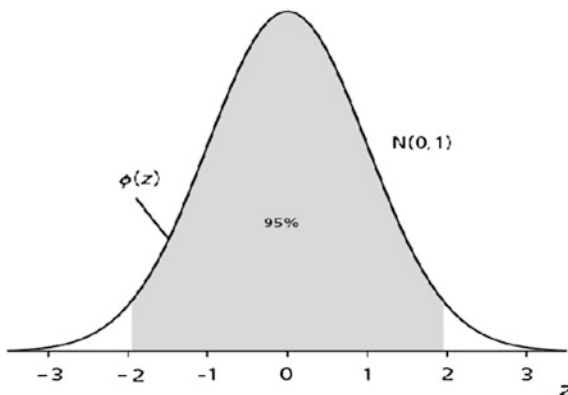
16.3.5 Bell-Shaped (Normal) Distribution

The German mathematician Gauss (Fig. 16.1) originally studied the *normal distribution*, and it is also known as the *Gaussian distribution*. It is shaped like a bell and so is popularly known as the *bell-shaped* distribution. The empirical frequencies of many natural populations exhibit a bell-shaped (normal) curve.

Fig. 16.1 Carl Friedrich Gauss



Fig. 16.2 Standard unit normal bell curve (Gaussian distribution)



The *normal distribution* N has mean μ and standard deviation σ . Its density function $f(x)$ where (where $-\infty < x < \infty$) is given by:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2} .$$

The *unit* (or *standard*) normal distribution $Z(0, 1)$ has mean 0 and standard deviation of 1 (Fig. 16.2). Every normal distribution may be converted to the unit normal distribution by $Z = (X - \mu)/\sigma$, and every probability statement about X has an equivalent probability statement about Z . The unit normal density function is given by:

$$f(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2}.$$

For a normal distribution, 68.2% of the data elements lie within one standard deviation of the mean; 95.4% of the population lies within two standard deviations of the mean; and 99.7% of the data lies within three standard deviations of the mean. For example, the shaded area under the curve within two standard deviations of the mean represents 95% of the population.

A fundamental result in probability theory is the *central limit theorem*, and this theorem essentially states that the sum of a large number of independent and identically distributed random variables has a distribution that is approximately normal. That is, suppose X_1, X_2, \dots, X_n is a sequence of independent random variables each with mean μ and variance σ^2 . Then, for large n , the distribution of

$$\frac{X_1 + X_2 + \dots + X_n - n\mu}{\sigma\sqrt{n}}$$

is approximately that of a unit normal variable Z . One application of the central limit theorem is in relation to the binomial random variables, where a binomial random variable with parameters (n, p) represents the number of successes of n independent trials, where each trial has a probability of p of success. This may be expressed as:

$$X = X_1 + X_2 + \dots + X_n,$$

where $X_i = 1$ if the i th trial is a success and is 0 otherwise. $E(X_i) = p$ and $\text{Var}(X_i) = p(1 - p)$, and then by applying the central limit theorem, it follows that for large n ,

$$\frac{X - np}{\sqrt{np(1 - p)}}$$

will be approximately a unit normal variable (which becomes more normal as n becomes larger).

The sum of independent normal random variables is normally distributed, and it can be shown that the sample average of X_1, X_2, \dots, X_n is normal, with a mean equal to the population mean but with a variance reduced by a factor of $1/n$.

$$E(\bar{X}) = \sum_{i=1}^n \frac{E(X_i)}{n} = \mu$$

$$\text{var}(\bar{X}) = \frac{1}{n^2} \sum_{i=1}^n \text{Var}(X_i) = \frac{\sigma^2}{n}$$

It follows that from this that the following is a unit normal random variable.

$$\sqrt{n} \frac{(X - \mu)}{\sigma}.$$

The term *six-sigma* (6σ) is a methodology concerned with continuous process improvement and aims for very high quality (close to perfection). A 6σ process is one in which 99.9996% of the products are expected to be free from defects (3.4 defects per million).

16.3.6 Frequency Tables, Histograms and Pie Charts

A frequency table is used to present or summarize data (Tables 16.4 and 16.5). It lists the data classes (or categories) in one column and the frequency of the category in another column.

A histogram is a way to represent data in bar chart format (Fig. 16.3). The data is divided into intervals, where an interval is a certain range of values. The horizontal axis of the histogram contains the intervals (also known as buckets), and the vertical axis shows the frequency (or relative frequency) of each interval. The bars represent the frequency, and there is no space between the bars.

A histogram has an associated shape. For example, it may resemble a normal distribution, a bi-modal or multi-modal distribution. It may be positively or negatively skewed. The construction of a histogram first involves the construction of a frequency table, where the data is divided into disjoint classes and the frequency of each class is determined.

A pie chart (Fig. 16.4) offers an alternate way to histograms in the presentation of data. A frequency table is first constructed, and the pie chart presents a visual representation of the percentage in each data class.

Table 16.4 Frequency table—Salary

Profession	Salary	Frequency
Project manager	65,000	3
Architect	65,000	1
Programmer	50,000	8
Tester	45,000	2
Director	90,000	1

Table 16.5 Frequency table—Test results

Mark	Frequency
0–24	3
25–49	10
50–74	15
75–100	2

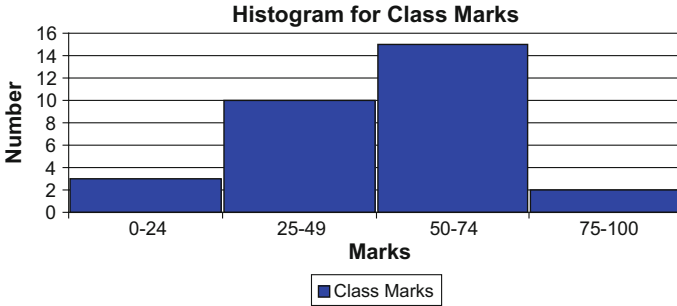


Fig. 16.3 Histogram test results

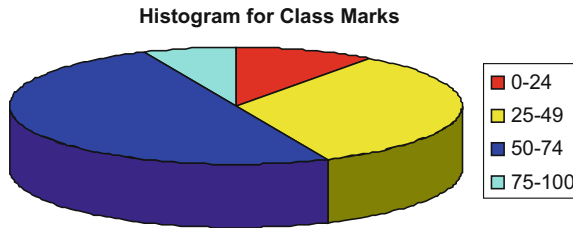


Fig. 16.4 Pie chart test results

16.3.7 Hypothesis Testing

The basic concept of inferential statistics is *hypothesis testing*, where a hypothesis is a statement about a particular population whose truth or falsity is unknown. Hypothesis testing is concerned with determining whether the values of the random sample from the population are consistent with the hypothesis. There are two mutually exclusive hypotheses: one of these is the null hypothesis H_0 and the other is the alternate research hypothesis H_1 . The null hypothesis H_0 is what the researcher is hoping to reject, and the research hypothesis H_1 is what the researcher is hoping to accept.

Statistical testing is then employed to test the hypothesis, and the result of the test is that we either reject the null hypothesis (and therefore accept the alternative hypothesis), or that we fail to reject (i.e. we accept) the null hypothesis. The rejection of the null hypothesis means that the null hypothesis is highly unlikely to be true, and that the research hypothesis should be accepted.

Statistical testing is conducted at a certain level of significance, with the probability of the null hypothesis H_0 being rejected when it is true never greater than α . The value α is called the level of significance of the test, with α usually being 0.1, 0.05, 0.005. A significance level β may also be applied to with respect to accepting the null hypothesis H_0 when H_0 is false.

Table 16.6 Hypothesis testing

Action	H_0 true, H_1 false	H_0 false, H_1 true
Reject H_1	Correct	False positive—Type 2 error $P(\text{Accept } H_0 H_0 \text{ false}) = \beta$
Reject H_0	False negative—Type 1 error $P(\text{Reject } H_0 H_0 \text{ true}) = \alpha$	Correct

The objective of a statistical test is not to determine whether or not H_0 is actually true, but rather to determine whether its validity is consistent with the observed data. That is, H_0 should only be rejected if the resultant data is very unlikely if H_0 is true.

The errors that can occur with hypothesis testing include type 1 and type 2 errors. Type 1 errors occur when we reject the null hypothesis when the null hypothesis is actually true. Type 2 errors occur when we accept the null hypothesis when the null hypothesis is false (Table 16.6).

For example, an example of a false positive is where the results of a blood test come back positive to indicate that a person has a particular disease when in fact the person does not have the disease. Similarly, an example of a false negative is where a blood test is negative indicating that a person does not have a particular disease when in fact the person does. Both errors can potentially be very serious.

The terms α and β represent the level of significance that will be accepted, and normally, $\alpha = \beta$. In other words, α is the probability that we will reject the null hypothesis when the null hypothesis is true, and β is the probability that we will accept the null hypothesis when the null hypothesis is false.

Testing a hypothesis at the $\alpha = 0.05$ level is equivalent to establishing a 95% confidence interval. For 99% confidence, α will be 0.01, and for 99.999% confidence, then α will be 0.00001.

The hypothesis may be concerned with testing a specific statement about the value of an unknown parameter θ of the population. This test is to be done at a certain level of significance, and the unknown parameter may, for example, be the mean or variance of the population. An estimator for the unknown parameter is determined, and the hypothesis that this is an accurate estimate is rejected if the random sample is not consistent with it. Otherwise, it is accepted.

The steps involved in hypothesis testing include:

1. Establish the null and alternative hypothesis.
2. Establish error levels (significance).
3. Compute the test statistics (often a t -test).
4. Decide on whether to accept or reject the null hypothesis.

The difference between the observed and expected test statistic, and whether the difference could be accounted for by normal sampling fluctuations is the key to the acceptance or rejection of the null hypothesis.

16.4 Review Questions

1. What is probability? What is statistics? Explain the difference between them.
2. Explain the laws of probability.
3. What is a sample space? What is an event?
4. Prove Boole's inequality $P(\cup_{i=1}^n E_i) \leq \sum_{i=1}^n P(E_i)$, where the E_i are not necessarily disjoint.
5. A couple has two children. What is the probability that both are girls if the eldest is a girl?
6. What is a random variable?
7. Explain the difference between the probability density function and the probability distribution function.
8. Explain expectation, variance, covariance and correlation.
9. Describe how statistics may be abused.
10. What is a random sample? Describe methods available to generate a random sample from a population. How may information be gained from a sample?
11. Explain how the average of a sample may be determined, and discuss the mean, mode and median of a sample.
12. Explain sample variance and sample standard deviation.
13. Describe the normal distribution and the central limit theorem.
14. Explain hypothesis testing and acceptance or rejection of the null hypothesis.

16.5 Summary

Statistics is an empirical science that is concerned with the collection, organization, analysis and interpretation and presentation of data. The data collection needs to be planned, and this may include surveys and experiments. Statistics is widely used by government and industrial organizations, and they may be used for forecasting as well as for presenting trends. Statistical sampling allows the behaviour of a random sample to be studied and inferences to be made about the population.

Probability theory provides a mathematical indication of the likelihood of an event occurring, and the probability is a numerical value between 0 and 1. A probability of 0 indicates that the event cannot occur, whereas a probability of 1

indicates that the event is guaranteed to occur. If the probability of an event is greater than 0.5, then this indicates that the event is more likely to occur than not to occur.

Statistical sampling is concerned with the methodology of choosing a random sample of a population, and the systematic study of the sample with the goal of drawing valid conclusions about the entire population.

Hypothesis testing is concerned with determining whether the values from a random sample from the population are consistent with a particular hypothesis. There are two mutually exclusive hypotheses: one of these is the null hypothesis H_0 and the other is the alternate research hypothesis H_1 . The null hypothesis H_0 is what the researcher is hoping to reject, and the research hypothesis H_1 is what the researcher is hoping to accept.

17.1 Introduction

Formal methods have been criticized for the limited availability of tools to support the software engineer in writing the formal specification and in conducting proof. Many of the early tools were criticized as not being of industrial strength. However, in recent years more advanced tools have become available to support the software engineer's work in formal specification and conducting proof, and this is likely to continue in the coming years.

The goal of this chapter is to give a flavour of a selection of tools¹ that are available to support the formal methodist in the performance of the various activities. Tools for VDM, Z, B, UML, model checking, and so on are considered. The approach is generally to choose tools to support the process, rather than choosing a process to support the tool.²

Mature organizations will generally employ a structured approach to the introduction of new tools. First, the requirements for a new tool are specified, and the options to satisfy the requirements are considered. These may include developing a tool internally; outsourcing the development of a tool, or purchasing a tool off the shelf. Finally, the users are trained on the tool, appropriate vendor support is provided, and the tool is rolled out throughout the organization. The vendor may need to provide dedicated support for a period of time postdeployment.

The tools include syntax checkers to determine whether the specification is syntactically correct; specialized editors which ensure that the written specification is syntactically correct; tools to support refinement; automated code generators that generate a high-level language corresponding to the specification; theorem provers to demonstrate the correctness of the refinement steps and to identify and resolve

¹The list of tools discussed in this chapter is intended to give a flavour of what tools are available, and the inclusion of a particular tool is not intended as a recommendation of that tool. Similarly, the omission of a particular tool should not be interpreted as disapproval of that tool.

²That is, the process normally comes first then the tool rather than the other way around.

theorem prover provides automated support with user support to direct the prover. Z/EVES supports almost the entire Z notation, and interaction with the tool is via its GUI. It runs on Linux and Windows and is used for commercial, academic and education purposes.

17.3 Tools for VDM

The IFAD Toolbox is a support tool for the VDM-SL specification language, and it provides support for syntax and type checking, an interpreter and debugger to execute and debug the specification, and a code generator to convert from VDM-SL to C++. It was originally developed by IFAD in Denmark, but they later sold the toolbox to CSK in Japan, and it was later renamed to VDMTools.

The Overture Integrated Development Environment (IDE) is an open-source tool for formal modelling and analysis of VDM-SL specifications (Fig. 17.2). It covers three dialects of VDM (VDM-SL, VDM++ and VDM-RT) [FiL:09], and it was developed by a community of volunteers on the Eclipse platform. The tool is intended to be industrial strength supporting practitioners in describing abstract models in software development, as well as supporting researchers in the formal methods community.

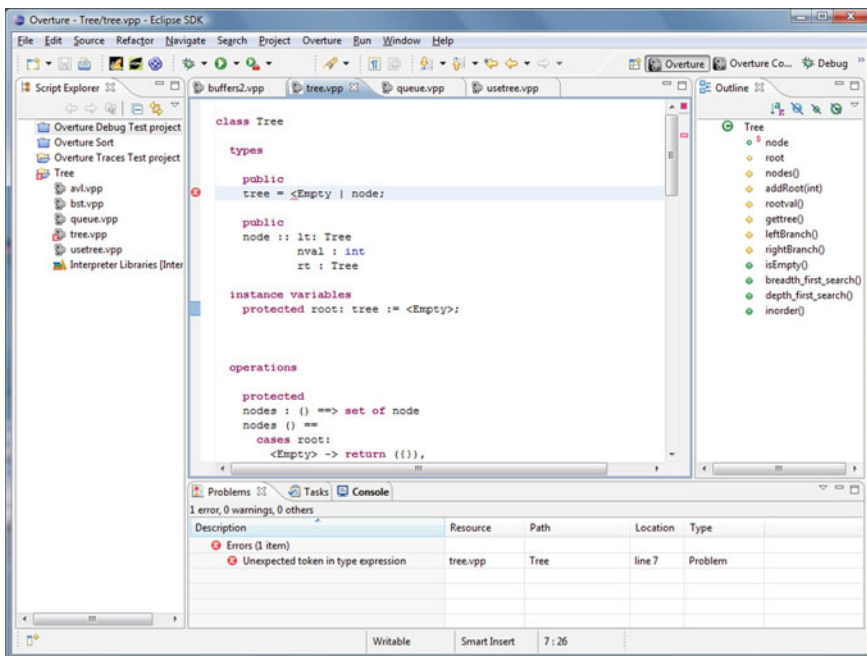


Fig. 17.2 Overture editor view

The tool may be downloaded from <http://overturetool.org>. It includes a large library of VDM-SL models that may be imported directly into the Overture tool. Overture can also work with the VDMTools as well as being a new open-source VDM tool set.

The tool provides functionality for syntax checking; an editor with syntax highlighting; proof obligation generator; interactive and automated proof support; model checking support; test generation support; code generation in C++ and Java; reverse engineering support; and UML visualization support.

17.4 Tools for B

There are a number of tools to support the B-Method including the B-Toolkit, Atelier B and the Rodin tool (for Event B). The *B-Toolkit* (from *B-Core*) is an integrated set of tools that supports the *B-Method*. It provides functionality for syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assistant and code generation. This, in theory, allows the complete formal development from the initial specification to the final implementation, with every proof obligation justified, leading to a provably correct program.

The abstract machine notation (AMN) is a state-based formal specification language (similar to Z or VDM), where an abstract machine consists of a state and operations on the state. The state is modelled by sets, relations and functions and the operations by pre- and postconditions using AMN notation.

Specification animation of the AMN specification is possible with the *B-Toolkit*, and this enables typical usage scenarios to be explored for requirements validation. This is, in effect, an early form of testing, and it may be used to demonstrate the presence or absence of desirable or undesirable behaviour. Verification takes the form of a proof to demonstrate that the invariant is preserved when the operation is executed within its precondition, and this is performed on the AMN specification with the *B-Toolkit*.

The *B-Method* and toolkit have been successfully applied in industrial applications, including the CICS project at IBM Hursley in the UK [Hoa:95]. The automated support provided has been cited as a major benefit of the application of the *B-Method* and the *B-Toolkit*. The *B-Toolkit* source code is now available.

The Atelier B tool supports formal specification and development in B, and it was developed by Clearsys (a French company that specializes in the development of safety critical software). There is a commercial version of the tool (the Maintenance Edition), as well as a freely available version (the Community Edition).

17.5 Tools for UML

There are many available tools for UML including IBM Rational Software Modeler® (RSM), which is a UML-based visual modelling and design tool (Fig. 17.3). It promotes communication and collaboration during design and development, and allows information about development projects to be specified and communicated from several perspectives. It is used for model-driven development and aligns the business needs with the product.

It gives the organization control over the evolving architecture and provides an integrated analysis and design platform. Abstract UML specifications may be built with traceability and impact analysis shown.

It has an intuitive user interface and a diagram editor to create expressive and interactive diagrams. The tool may be integrated with other IBM Rational tools such as Clearcase, Clearquest and Requisite Pro.

IBM Rational Rhapsody® is a visual development environment used in real-time or embedded systems. It helps teams collaborate to understand and elaborate requirements; abstract complexity using modelling languages such as UML; validate functionality early in development; and automate code generation to speed up the development process.

Sparx Enterprise Architect is a UML analysis and design tool used for modelling business and IT systems. It covers the full product development lifecycle and supports automated document generation, code generation and reverse engineering

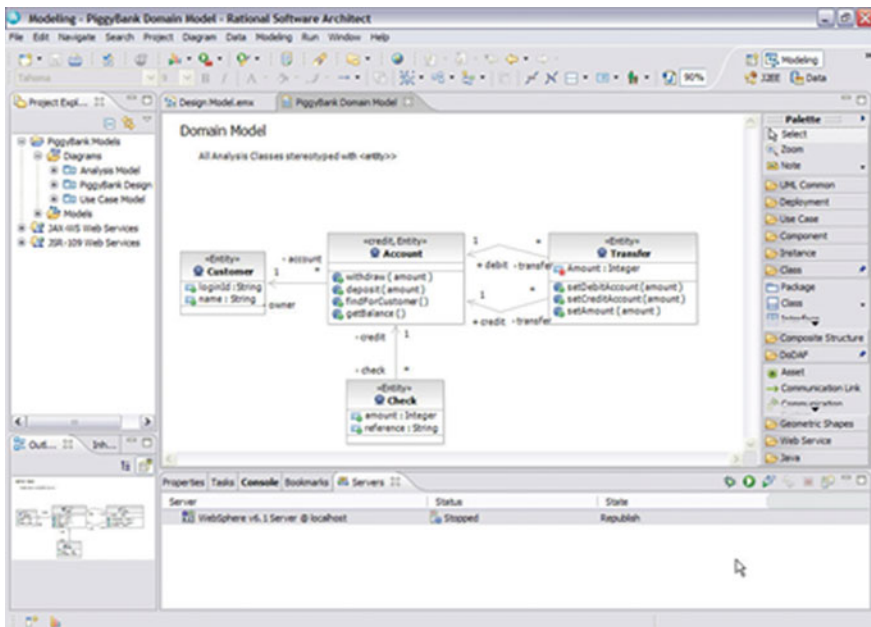


Fig. 17.3 IBM rational software modeler

of source code. Its reverse engineering feature allows a visual representation of the software application to be provided.

It can model, manage and trace requirements to the design, test cases and deployment, and it can trace the implementation of the system requirements to model elements. It can search and report on requirements and perform an impact analysis on proposed changes to the requirements.

17.6 Tools for Model Checking

Model checking extracts a finite model from a system and then checks some property of that model. The model checker performs an exhaustive state search (i.e. it checks all system states to determine whether they satisfy the desired property or not). If a state that violates the desired property is determined (i.e. a defect has been found once it is shown that the system does not fulfil one of its specified properties), then the model checker provides a counter-example indicating how the model can reach this undesired state. The system is considered to be correct if it satisfies all of the specified properties.

There are various tools for model checking including Spin, Bandera, SMV and UppAal. The Spin tool was developed at Bell Labs in the early 1980s, and it checks finite-state systems with properties specified by linear temporal logic (LTL) to determine if the property is always satisfied. It generates a counter-example trace if determines that a property is violated.

Bandera is a tool for model checking Java source code, and it automates the extraction of a finite-state model from the Java source code. It then translates into an existing model checker's input language. The properties to be verified are specified in the Bandera Specification Language (BSL).

17.7 Tools for Theorem Provers

There are many automated and interactive theorem provers including the Boyer-Moore theorem prover, ACL2, Otter, PVS, Theorem Proving System and HOL. We discussed a sample of these tools in Chap. 15, and in this section, we focus on PVS and HOL. PVS was developed at the computer laboratory at SRI in the USA, and HOL was developed at Cambridge University in the UK.

The Prototype Verification System (PVS) is a mechanized environment for formal specification and verification. It includes a specification language integrated with support tools and an interactive theorem prover. The specification language is based on *higher-order logic*, and the theorem prover is interactive and guided by the user in conducting proof. PVS has been applied to the verification of hardware and software and has been used by NASA as part of the Space Shuttle flight control requirements specification.

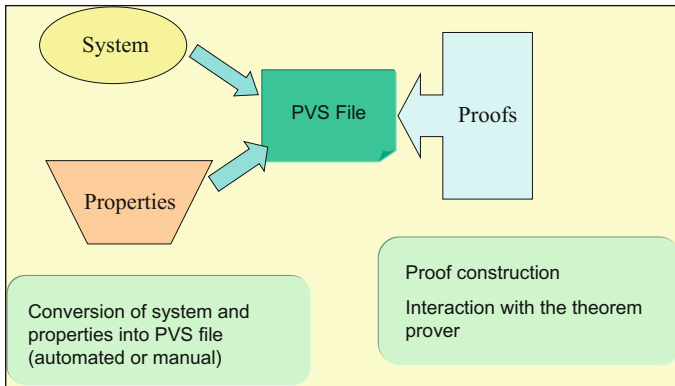


Fig. 17.4 PVS system

PVS (Fig. 17.4) is implemented in Common Lisp and is available under the GNU General Public License. It is a large and complex system and requires significant investment in time to become a moderately skilled user.

The PVS language is close to “normal” notation, and the language includes types, type checking, recursion, lambda notation and abstract data types. There are essentially two languages in PVS: a language to write definitions (definition language), and a language to prove theorems (proof language). The PVS theorem prover has a built-in model checker.

The HOL system was developed at Cambridge University, and it is an environment for interactive theorem proving in higher-order logic. It requires skilled human guidance and has been used for the verification of microprocessor design. It is a widely used theorem prover.

17.8 Review Questions

1. What is the purpose of tool support for formal methods?
2. What tools can support Z? Compare and contrast.
3. What tools can support VDM? Compare and contrast.
4. What tools can support UML? Compare and contrast.
5. What tools can support B? Compare and contrast.
6. What tools can support automated and interactive theorem provers?
7. What tools can support model checkers? Compare and contrast.

17.9 Summary

The goal of this chapter was to give a flavour of a selection of tools that are available to support the formal methodist in the performance of the various activities. Tools for VDM, Z, B, UML, theorem provers and model checking were considered. Many of the early tools to support formal methods were criticized as not being of industrial strength. However, in recent years more advanced tools have become available to support the software engineer's work in formal specification and formal proof, and this will continue in the coming years.

We discussed the *ZEVES* tool which allows *Z* specifications to be entered, edited and analysed in their typeset form. It supports the analysis of specifications, and its theorem prover provides automated support with user guidance employed to direct the prover.

We discussed the Overture Integrated Development Environment, which is an open-source tool for formal modelling and analysis of VDM-SL specifications. It was developed by a community of volunteers on the Eclipse platform, and the tool is intended to be of industrial strength in supporting practitioners in describing abstract models in software development.

We discussed the *B-Toolkit* (from *B-Core*) which is an integrated set of tools that supports the *B-Method*. It provides functionality for syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assister and code generation. This, in theory, allows the complete formal development from the initial specification to the final implementation, with every proof obligation justified, leading to a provably correct program.

18.1 Introduction

Technology transfer is concerned with the practical exploitation of new technology developed by an academic or industrial research group, and the objective is to facilitate the use of the technology in an industrial environment. The transfer of new technology and leading edge methods to an industrial environment needs to take place in a controlled manner. It cannot be assumed that a new technology or method will necessarily suit an organization, and the initial focus is concerned with piloting the new technology, and measuring the benefits gained from its use.

The pilot(s) will provide insight into the effectiveness of the new technology, as well as identifying areas that require further improvement prior to general deployment. Feedback from the pilot is provided to the research groups, and further improvements and pilots may take place as appropriate. In some instances, it may be that commercial exploitation is inappropriate. This may be due to immaturity of the technology, the fact that it does not suit the culture of the company, or the fact that the evaluation of the technology did not achieve the required criteria, and is not expected to in the near future.

A pilot of new technology is an experiment to determine its effectiveness and to enable an informed decision to be made on the benefits of transferring the new technology throughout the company. The pilot needs to be planned and this includes deciding which people will participate, the provision of training for the participants and the identification of objective criteria to evaluate the new technology. The objective criteria are identified prior to the commencement of the pilot, and the results of the pilot are then compared to the evaluation criteria. Further analysis of the evaluation results then takes place, and this allows an informed decision to be made as to whether it is appropriate to deploy the technology throughout the company.

Organization culture needs to be considered for effective technology transfer. Every organization has a distinct culture and this is reflected in the way that people work, and in the way in which things are done. Organization culture includes the

ethos and core values of the organization, its commitment or resistance to change, and so on. The transfer of new technologies will be easier in cultures where there are an openness and willingness to change. However, in other cultures the deployment of new technology may be difficult due to resistance to change within the organization.

18.2 Formal Methods and Industry

The formal methods community have developed many elegant formalisms to assist the development of high-quality software. However, in practice many software developers find formal methods difficult to use, whereas the formal methods community seem unable to understand why software developers find their notation and methods difficult. The chasm between the formal methods community and industrial programmers has been considered in [Web:93], and various characteristics of a good formal method considered.

It is not, of course, the role of the formal methods community to sell formal methods to industry: rather, their role is to develop notations and methods that are useful and usable; provide education on applying formal methods to students and interested industrialists and to provide expert support during pilots of formal methods in an industrial environment. However, in order to develop a usable formal method, it is essential that the formal methods community has a better understanding of the needs of industry and the commercial constraints of industrial companies and projects.

An industrial project consists of a project team, and each team member has various responsibilities in the software development process (e.g. requirements, definition, design, implementation, software testing, configuration management, project management). A project is subjected to strict budget, timeliness and quality constraints, and the project manager is responsible for delivering a high-quality product on time and budget to the customer. An industrial project follows a defined software process, and there is a need to define the process to be followed when formal methods are to be part of the process.

Formal methods need to be piloted prior to their general deployment in an organization to ensure that there are real benefits gained in higher quality software from their use, and that the commercial constraints (e.g. budget, timeliness and quality) are addressed. Late projects can cause the loss of a market opportunity or cause major customer dissatisfaction and a loss of credibility. Budget overruns and quality problems can lead to financial loss to the company.

Industry is generally concerned with finding the most cost-effective solution to delivering high-quality software on time and within budget. The natural question [Wic:00] is “*Under what circumstances does formal methods provide the most cost*

effective solution to an industrial problem".¹ Any selling of formal methods to industry must be realistic, as an overselling of the benefits of formal methods has led to a negative perception of the technology. This is since industrialists have experienced difficulties in working with the immaturity of formal methods, and in practice formal methods have not been used extensively due to problems with their usability. One infamous overselling of formal methods was the verification of the VIPER microprocessor, with RSRE and Charter overselling VIPER as a chip design that conforms to its formal specification [Tie:91].

The development of standards such as 00-55 (British Defense Standard for the procurement of safety critical software), DO-178B (US standard for certification of safety critical avionics software) and IEC 61509 (international standard for critical systems) all mention formal methods as a way to assure high-quality software. The 00-55 Defense Standard initially mandated the use of formal methods for formal code verification and specification animation.

The safety critical field is one area where the use of formal methods has shown good benefits, and where the verification of correctness is essential. Quality and safety cannot be comprised in safety critical systems, and the regulated sector has provided some good case studies in the application of formal methods. These include the Darlington Nuclear Generation Station in Canada, where formal methods were used to certify the shutdown software of the plant. Other applications include the use of formal methods to certify the correctness of safety critical software is the Paris metro signaling system. The regulatory sector is concerned with certification of the code with respect to the requirements, and timeliness is less important than the actual certification. Regulatory systems are generally expensive as often there are several organizations and products involved, and the verification of the correctness of these products is time consuming.

The application of formal methods to mainstream software engineering has been less successful. Mainstream software engineering is subjected to stringent commercial constraints, and the experience to date is that the use of formal methods does not provide any appreciable gain in quality, timeliness or productivity.² Time to market is often a key commercial driver in mainstream software engineering.

¹The answer at this time is that the applications of formal methods to the regulated sector bring benefits, as this sector requires certification that the code meets stringent safety critical and security critical requirements. The cost of certification in the regulated sector is high, but formal methods can be employed to demonstrate to regulators that the code conforms to the requirements. The benefit gained is the extra quality assurance gained from the use of formal methods. The reliability requirements in the regulated sector are much higher than for conventional systems, and the cost of testing may make it infeasible to verify that these reliability requirements have been achieved. This makes the use of formal methods a cost-effective solution in this domain. The demonstration includes mathematical proof and testing. As the maturity of formal methods evolves there may be other circumstances in which formal methods provide the most cost-effective solution.

²The CICS project at IBM claimed a 9% increase in productivity. However, the late Peter Lupton of IBM outlined difficulties that the engineers had with formal specification in Z at the FME'93 conference. Care needs to be taken with measurements in software engineering as some software metrics are unsound. For example, it is easy to increase the productivity of an organization (income per employee) with a redundancy program. Programmer productivity in terms of lines of

18.3 Usability of Formal Methods

There are practical difficulties associated with the usability of formal methods. It seems to be assumed that programmers and even customers are willing to become familiar with the mathematics used in formal methods. There is little evidence to suggest that customers in mainstream software engineering would be prepared to use formal methods.³ Customers are concerned with their own domain and speak the technical language of that domain.⁴ Often, the use of mathematics is an alien activity that bears little resemblance to their normal practical work. Programmers are interested in programming rather than in mathematics and generally have no interest in becoming mathematicians.⁵

However, the mathematics involved in most formal methods is reasonably elementary, and, in theory, if both customers and programmers are willing to learn the formal mathematical notation, then a rigorous validation of the formal specification can take place to verify its correctness. Both parties can review the formal specification to verify its correctness, and the code can be verified to be correct with respect to the formal specification. It is usually possible to get a developer to learn a formal method, as a programmer has some experience of mathematics and logic; however, in practice, it is more difficult to get a customer to learn a formal method.

This means that often a formal specification of the requirements and an informal definition of the requirements using a natural language are maintained. It is essential that both of these documents are consistent and that there is a rigorous validation of the formal specification. Otherwise, if the programmer proves the correctness of the code with respect to the formal specification, and the formal specification is incorrect, then the formal development of the software is incorrect. There are several techniques to validate a formal specification (Table 18.1), and these are described in more detail in [Wic:00].

18.3.1 Why Are Formal Methods Difficult?

Formal methods are perceived as being difficult to use and of offering limited value in mainstream software engineering. Programmers receive some training in mathematics as part of their education. However, in practice, most programmers who

code per week is also unsound as a measure of productivity as the quality of the code also needs to be considered.

³The domain in which the software is being used will influence the willingness or otherwise of the customers to become familiar with the mathematics required. Certainly, in mainstream software engineering there is little interest from customers, and the perception is that formal methods are unusable. However, in some domains such as the regulated sector there is a greater willingness of customers to become familiar with the mathematical notation.

⁴Most customers have a very limited interest and even less willingness to use mathematics (exception to this are the regulated sector).

⁵Mathematics that is potentially useful to software engineers is discussed in [ORg:12, ORg:16b].

Table 18.1 Techniques for validation of formal specification

Technique	Description
Proof	This involves demonstrating that the formal specification adheres to key properties of the requirements. The implementation will need to preserve these properties also
Software inspections	This involves a Fagan like inspection (discussed in Chap. 1) to perform the validation. It may involve comparing an informal set of requirements (unless the customer has learned the formal method) with the formal specification
Specification animation	This involves program (or specification) execution as a way to validate the formal specification. It is similar to testing
Tools	Tools provide some limited support in validating a formal specification

Table 18.2 Factors in difficulty of formal methods

Factor	Description
Notation/intuition	The notation employed differs from that employed in classical mathematics. Intuition varies from person to person, but many programmers find the notation in formal methods to be unintuitive
Formal specification	It is easier to read a formal specification than to write one
Validation of formal specification	The validation of a formal specification using proof techniques or a Fagan like inspection is difficult
Refinement ^a	The refinement of a formal specification into more concrete specifications, with the proof of validity of each refinement step is difficult and time consuming
Proof	Proof can be difficult and time consuming
Tool support	Many of the existing tools are difficult to use

^aThe author doubts that refinement is cost effective for mainstream software engineering. However, it may be useful in the regulated environment

learn formal methods at university never use formal methods again once they take an industrial position.

It may well be that the very nature of formal methods is such that it is suited only for specialists with a strong background in mathematics. Some of the reasons why formal methods are perceived as being difficult are listed in Table 18.2.

18.3.2 Characteristics of a Usable Formal Method

It is important to investigate ways by which formal methods can be made more usable to software engineers. This may involve designing more usable notations and better tools to support the process. Education and training is important, and bringing in an expert formal methods consultant to act as a mentor to the team can be beneficial. The consultant can educate the team on reading and writing formal specifications, as well as providing mentoring support. The consultant can give

Table 18.3 Characteristics of a usable formal method

Characteristic	Description
Intuitive	A good intuitive notation has potential as a usable formal method. Intuition varies among people
Teachable	A formal method needs to be teachable to the average software engineer. The training should include (at least) writing practical formal specifications
Tool support	Usable tools to support formal specification, validation, refinement and proof are required
Adaptable to change	Change is common in a software engineering environment. A usable formal method should be adaptable to change
Technology transfer path	The process for software development needs to be defined to include formal methods. The migration to formal methods needs to be managed
Cost ^a	The use of formal methods should be a cost effective (timeliness, quality and productivity). There should be a return on investment from the use of formal methods

^aA commercial company will expect a return on investment from the use of a new technology. This may be reduced software development costs, improved quality, improved timeliness of projects or improvements in productivity. A company does not go to the trouble of deploying a new technology just to satisfy academic interest

practical advice on choosing the formal specification language (e.g. VDM, Z, B, etc.), appropriate tool support including theorem provers, and to develop knowledge of formal specification and verification in the company. Some of the characteristics of a usable formal method are suggested in Table 18.3.

18.4 Pilot of Formal Methods

The transfer of new technology to the organization involves a structured pilot of the new technology using objective evaluation criteria. A decision is made following the pilot to either conduct further pilots, abandon the technology or to transfer the technology within the company. The steps for a pilot of formal methods are listed in Table 18.4.

18.4.1 Technology Transfer of Formal Methods

The transfer of new technology to an organization needs to be done in a controlled manner. The steps in the technology transfer are shown in Table 18.5.

Table 18.4 Steps for pilot of formal methods

Step	Description
Overview of technology	This provides the motivation for the pilot of the technology. An organization (or group) receives an overview of a new technology that offers potential. For example, this may be an approach such as Z or VDM
Select pilot project	The technology may be sufficiently promising for the organization to consider a pilot. This involves identifying a suitable project for the pilot and selecting the project participants
Process for pilot	Define the project's software process to be followed for the pilot. The process will detail where formal methods will be used in the lifecycle
Training	Provide training on the new technology (formal method) and the process for the pilot. The training will require the students to write formal specifications
Evaluation criteria (Pilot)	Define objective criteria to judge the effectiveness of the new technology. This includes gathering data for: <ul style="list-style-type: none"> • Productivity Measurements • Quality Measurements • Timeliness Measurements
Support (Pilot)	Provide on-site support to assist the developers in preparing formal specifications. This may require consultants
Conduct pilot	The pilot is conducted and the coordinator for the pilot will work to address any issues that arise. Data is gathered to enable objective evaluation to take place
Post-mortem ^a	A post-mortem is conducted after the pilot to consider what went well and what went poorly. The evaluation criteria are compared against the gathered data and recommendations are made to either conduct further pilots, abandon the technology or to institutionalize the new technology

^aIt may well be that the result of a pilot of formal methods results in a decision that the methodology is inappropriate for the company at this time. The bottom line is whether formal methods provide a more cost-effective solution to software engineering problems than other engineering approaches. Further pilots may be required before a final decision can be made

Table 18.5 Steps for technology transfer of formal methods

Step	Description
Decision to deploy	A decision is made to transfer the new technology throughout the organization. The results of the pilot justify the decision
Software development process	Update the software development process to define how formal methods are used as part of the development process
Training	Provide practical training to all affected staff in the company. The training will include writing formal specifications
Audits	Verify that the new process is being followed and that it is effective by conducting audits. The results of the audits are reported in management
Improvements	Potential improvements to the technology or process are identified and acted upon

18.5 Review Questions

1. Why are formal methods perceived as being difficult by many industrialists?
2. What is the purpose of a pilot of formal methods? Describe the steps involved.
3. What are the characteristics of a usable formal method?
4. What are the steps involved in technology transfer of formal methods to an organization?
5. What techniques are employed to validate a formal specification?

18.6 Summary

Technology transfer is the disciplined transfer of new technology to a company and is concerned with the practical exploitation of the new technology. It cannot be assumed that a new technology or method will necessarily benefit an organization, and the initial focus is concerned with piloting the new technology and measuring the benefits gained from its use.

The pilot(s) will provide insight into the effectiveness of the new technology, as well as identifying areas that require further improvement prior to general deployment of the technology throughout the company.

The pilot is generally limited to one part of the company or to one specific project in the company. A pilot needs to be planned and this includes deciding who will participate, the provision of training for the participants and the identification of criteria to evaluate the new technology.

The results of the pilot are then compared to the evaluation criteria and further analysis and a post-mortem take place. This allows an informed decision to be made as to whether the deployment of the new technology throughout the entire company is appropriate.

We embarked on a long journey in this book and set ourselves the objective of providing a concise introduction to the formal methods field to students and practitioners. The goal was to cover both theory and practice, and to give the reader a grasp of the fundamentals of the formal methods field, as well as guidance on how to apply the theory in an industrial environment.

We noted that companies today need to focus on customer satisfaction and on software quality, and need to ensure that the desired quality is built into the software product. Customers today have very high expectations on quality, and expect high-quality software products to be consistently delivered on time. The focus on quality requires that the organization defines a sound software development infrastructure to enable quality software to be consistently produced.

Quality improvement also requires that the organization be actively aware of industrial best practice, as well as emerging technologies from various research programs. Piloting or technology transfer of innovative technology is an important part of continuous improvement. Formal methods are one innovative technology that may be employed to provide additional confidence in the correctness of the software, and it has a role to play in software engineering (especially in the safety and security critical fields).

We started our journey with a discussion of approaches used in current software engineering to build quality into software. We discussed software project management; software processes and software life cycles; software inspections and testing; and the Agile methodology. We discussed software process improvement using the CMMI, and we noted that it provides a framework to assess the current capability or maturity of selected software processes and to prioritize improvements.

We then discussed software reliability and dependability, and covered topics such as software reliability and software reliability models; the Cleanroom methodology, system availability, safety and security critical systems, and dependency engineering.

We discussed formal methods, which consist of a set of mathematic techniques that provide an extra level of confidence in the correctness of the software. They may be employed to formally state the requirements of the proposed system, and to derive a program from its mathematical specification. They may be employed to provide a rigorous proof that the implemented program satisfies its specification. They have been mainly applied to the safety critical field.

We then provided an introduction to fundamental building blocks in discrete mathematics including sets, relations and functions. A set is a collection of well-defined objects, and it may be finite or infinite. A relation between two sets A and B indicates a relationship between members of the two sets, and is a subset of the Cartesian product of the two sets. A function is a special type of relation such that for each element in A there is at most one element in the co-domain B . Functions may be partial or total and injective, surjective or bijective.

We then presented a short history of logic, and we discussed Greek contributions to syllogistic logic, stoic logic, fallacies and paradoxes. Boole's symbolic logic and its application to digital computing were discussed, and we considered Frege's work on predicate logic.

We then provided an introduction to propositional and predicate logic. Propositional logic may be used to encode simple arguments that are expressed in natural language, and to determine their validity. The nature of mathematical proof was discussed, and we presented proof by truth tables, semantic tableaux and natural deduction. Predicate logic allows complex facts about the world to be represented, and new facts may be determined via deductive reasoning. Predicate calculus includes predicates, variables and quantifiers, and a predicate is a characteristic or property that the subject of a statement can have.

We then presented some advanced topics in logic including fuzzy logic, temporal logic, intuitionistic logic, dealing with undefined values, theorem provers and the applications of logic to AI. Fuzzy logic is an extension of classical logic that acts as a mathematical model for vagueness. Temporal logic is concerned with the expression of properties that have time dependencies, and it allows properties about the past, present and future to be expressed. Intuitionism was a controversial theory on the foundations of mathematics based on a rejection of the law of the excluded middle, and an insistence on constructive existence. We discussed three approaches to deal with undefined values, including the logic of partial functions; Dijkstra's approach with his *cand* and *cor* operators; and Parnas's approach which preserves a classical two-valued logic.

We discussed the Z specification language, which is one of the more popular formal methods. It was developed at the Programming Research Group at Oxford University in the early 1980s. Z specifications are mathematical, and the use of mathematics ensures precision, and allows inconsistencies and gaps in the specification to be identified. Theorem provers may be employed to demonstrate that the software implementation meets its specification.

We presented the Vienna Development Method, which is one of the more popular formal specification languages. We described the history of its development at IBM in Vienna, the main features of the language and its development method. We discussed a variant termed the Irish school of VDM (VDM⁺) and explained how it differs from standard VDM.

We then discussed the unified modelling language (UML), which is a visual modelling language for software systems, and used to present several views of the system architecture. It was developed at Rational Corporation as a notation for modelling object-oriented systems. We presented various UML diagrams such as use case diagrams, sequence diagrams and activity diagrams.

We then considered the approach of Dijkstra, Hoare and Parnas. We discussed the calculus of weakest preconditions developed by Dijkstra and the axiomatic semantics of programming languages developed by Hoare. We then discussed the classical engineering approach of Parnas.

We discussed automata theory, including finite-state machines, pushdown automata and Turing machines. Finite-state machines are abstract machines that are in only one state at a time, and the input symbol causes a transition from the current state to the next state. Pushdown automata have greater computational power, and they contain extra memory in the form of a stack from which symbols may be pushed or popped. The Turing machine is the most powerful model for computation, and is equivalent to an actual computer in the sense that it can compute exactly the same set of functions.

We then discussed model checking which is an automated technique such that given a finite-state model of a system and a formal property, then it systematically checks whether the property is true or false in a given state in the model. It is an effective technique to identify potential design errors, and it increases the confidence in the correctness of the system design.

We then discussed the nature of proof and theorem provers including automated and interactive theorem provers. We discussed the nature of a mathematical proof and a formal mathematical proof.

We discussed probability and statistics including a discussion on discrete random variables; probability distributions; sample spaces; sampling; the abuse of statistics; variance and standard deviation; and hypothesis testing.

We then discussed a selection of tools that are available to support the formal methodist in the performance of the various activities. Tools for VDM, Z, B, UML, theorem provers and model checking were considered.

Finally, we discussed technology transfer which is concerned with the practical exploitation of new technology developed by an academic or industrial research group, and the objective is to facilitate the use of the technology in an industrial environment.

19.1 The Future of Formal Methods

Quality is fundamental to the success of a company, and there will be a continued focus on quality (e.g. approved quality systems such as ISO 9001, or achieving a specific CMMI maturity level). Customer expectations are increasing all the time and expect a high-quality product to be consistently delivered.

The safety and security critical fields will continue to demand extra assurance of the quality and reliability of software, and formal methods will continue to play a key role. Software components and the verification of software components may become increasingly important, as companies will wish to speed up development to shorten the time to market. Formal methods may play a role in the verification of software components.

The debate in relation to the use of mathematics in software engineering is ongoing. Many practitioners are against their use and employ methodologies such as software inspections and testing to build quality into the software. They argue that the use of mathematical techniques would seriously impact the market opportunity. Industrialists often need to balance conflicting needs such as quality, cost and delivering on time. They argue that the commercial realities require methodologies and techniques that allow them to achieve their business goals effectively.

The other camp argues that the use of mathematics is essential in the delivery of high-quality and reliable software, and that if a company does not place sufficient emphasis on quality, it will pay the price in terms of poor quality and loss of reputation.

It is generally accepted that mathematics and formal methods must play a role in the safety critical and security critical fields. Apart from that, the extent of its use remains a hotly disputed topic. It is unrealistic to expect companies to deploy formal methods unless they have evidence that it will support the delivery of high-quality products to the market place ahead of their competitors. Formal methods need to prove that it can do this if it wishes to be taken seriously in mainstream software engineering.

Formal specification languages will continue to evolve and become more usable. There will be more usable theorem provers and tools to support the formal methodist in delivering high-quality software.

References

Ack:94

J.L. Ackrill, *Aristotle the Philosopher* (Clarendon Press Oxford, 1994)

Ada:84

E. Adams, Optimizing preventive service of software products. *IBM Res. J.* **28**(1), 2–14 (1984)

BaK:08

C. Baier, J.P. Katoen, *Principles of Model Checking* (MIT Press, Cambridge, MA) 28

Bec:

K. Beck, *Extreme Programming Explained. Embrace Change*, vol 20 (Addison Wesley, Reading)

BjJ:78

D. Bjørner, C. Jones, in *The Vienna Development Method. The Meta language*. Lecture Notes in Computer Science, vol 61 (Springer, Berlin, 1978)

BjJ:82

D. Bjørner, C. Jones, in *Formal Specification and Software Development*. Prentice Hall International Series in Computer Science (1982)

Boe:88

B. Boehm, A spiral model for software development and enhancement. *Computer* (May 1988)

BoM:79

R. Boyer, J.S. Moore, *A Computational Logic. The Boyer Moore Theorem Prover* (Academic Press, New York, 1979)

Boo:48

G. Boole, The calculus of logic. *Camb. Dublin Math. J.* **III**, 183–98 (1848)

Boo:58

G. Boole, *An Investigation into the Laws of Thought* (Dover Publications, 1958) (First published in 1854)

Brk:75

F. Brooks, *The Mythical Man Month* (Addison Wesley, Reading, 1975)

Brk:86

F. Brooks, No silver bullet. Essence and accidents of software engineering, in *Information Processing* (Elsevier, Amsterdam, 1986)

Bro:90

M.J.D Brown, Rational for the development of the U.K. Defence Standards for Safety Critical software, in Compass Conference (1990)

But:00

VDM^{*} Mathematical Structures for Formal Methods. Presentation by Andrew Butterfield. Foundations and Methods Group. Trinity College, Dublin, 19th May 2000

CKS:11

M.B. Chrissis, M. Conrad, S. Shrum, *CMMI Guidelines for Process Integration and Product Improvement*, 3rd edn. SEI Series in Software Engineering (Addison Wesley, Reading, 2011)

CM:81

E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in *Logic of Programs: Work-shop*, Yorktown Heights, NY, May 1981, volume 131 of LNCS (Springer, Berlin, 1981)

Cod:70

E.F. Codd, A relational model of data for large shared data banks. *Commun. ACM* **13**(6), 377–387 (1970)

CoM:90

R.H. Cobb, H.D. Mills, Engineering software under statistical quality control. *IEEE Softw.* (1990)

Dat:81

C.J. Date, in *An Introduction to Database Systems*. 3rd edn. The Systems Programming Series (1981)

Dij:76

E.W. Dijkstra, *A Discipline of Programming* (Prentice Hall, Englewood Cliffs, NJ, 1976)

Dil:90

A. Diller, Z. *An Introduction to Formal Methods* (Wiley, England, 1990)

Fag:76

M. Fagan, Design and code inspections to reduce errors in software development. *IBM Syst. J.* **15** (3) (1976)

FiL:09

J. Fitzgerald, P.G. Larsen, *Modelling Systems—Practical Tools and Techniques in Software Development*, 2nd edn, vol 29 (Cambridge University Press, Cambridge)

Flo:67

R. Floyd, Assigning Meanings to Programs, in *Proc. Symp. Appl. Math.* (19), 19–32 (1967)

Ger:94

S. Gerhart, D. Craighen, T. Ralston, Experience with formal methods in critical systems. *IEEE Softw.* (January 1994)

Gri:81

D. Gries, *The Science of Programming* (Springer, Berlin, 1981)

HB:95

M. Hinchey, J. Bowen (eds.), in *Applications of Formal Methods*. Prentice Hall International Series in Computer Science (1995)

Hey:66

A. Heyting, *Intuitionist Logic. An Introduction* (North-Holland Publishing, 1966)

HoA:95

J.P. Hoare, Application of the B method to CICS, in *Applications of Formal Methods*. Prentice Hall International Series in Computer Science (1995)

Hor:69

C.A.R. Hoare, An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–585 (1969)

Hor:85

C.A.R. Hoare, in *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science (1985)

HoU:79

J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Boston (1979)

InA:91

D. Andrews, D. Ince, *Practical Formal Methods with VDM* (McGraw Hill International, 1991)

Jac:99a

I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Modeling Language User Guide* (Addison-Wesley, Reading, 1999a)

Jac:99b

I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process* (Addison-Wesley, Reading, 1999b)

Jan:97

R. Janicki, On a Formal Semantics of Tabular Expressions. Technical Report CRL 355. Communications Research Laboratory, McMaster University, Ontario (1997)

Jon:86

C. Jones, *Systematic Software Development using VDM* (Prentice Hall International, 1986)

Kel:97

J. Kelly, *The Essence of Logic* (Prentice Hall, Englewood Cliffs NJ, 1997)

Kuh:70

T. Kuhn, *The Structure of Scientific Revolutions* (University of Chicago Press, Chicago, 1970)

Lak:76

I. Lakatos, *Proof and Refutations. The Logic of Mathematical Discovery* (Cambridge University Press, Cambridge, 1976)

Lof:84

P. Martin Löf, *Intuitionist Type Theory*. Notes by Giovanni Savin of lectures given in Padua, June, 1980. Bibliopolis. Napoli (1984)

Mac:90

M. Mac An Airchinnigh, *Computation Models and Computing*, Ph.D. Thesis, Department of Computer Science, Trinity College Dublin

MacK:95

D. MacKensie, The automation of proof. *IEEE Hist. Sociol. Explor. Ann. Hist. Comput.* **17**(3) (1995)

Mc:59

J. McCarthy, Programs with common sense, in *Proceedings of the Teddington Conference on the Mechanization of Thought Processes* (1959)

McD:94

E. McDonnell, M.Sc. Thesis. Department of Computer Science, Trinity College Dublin

McH:85

D. McHale, *Boole* (Cork University Press, 1985)

Men:87

E. Mendelson, *Introduction to Mathematical Logic* (Wadsworth and Cole/Brook, Advanced Books & Software, 1987)

Mil:89

R. Milner et al., in *A Calculus of Mobile Processes*. Part 1. LFCS Report Series. ECS-LFCS-89-85. Department of Computer Science, University of Edinburgh

MOD:91a

UK Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment*. Part 1: Requirements. Interim Defence Standard -55 (Part 1)/Issue 1 (1991a)

MOD:91b

UK Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment*. Part 2: Guidance. Interim Defence Standard -55 (Part 2)/Issue 1 (1991b)

NeS:56

A. Newell, H. Simon, The logic theory machine. *IRE Trans. Inf Theory* **2**, 61–79 (1956)

ORg:06

G. O' Regan, *Mathematical Approaches to Software Quality*, vol 26 (Springer, London)

ORg:10

G. O' Regan, *Introduction to Software Process Improvement*. (Springer, London, 2010)

ORg:12

G. O' Regan, *Mathematics in Computing*, 2nd edn (Springer, London, 2012)

ORg:13

G. O' Regan, *Giants of Computing* (Springer, London, 2013)

ORg:14

G. O' Regan, *Introduction to Software Quality* (Springer, Switzerland, 2014)

ORg:16a

G. O' Regan, *Introduction to the History of Computing* (Springer, Switzerland, 2016a)

ORg:16b

G. O' Regan, *Guide to Discrete Mathematics* (Springer, Switzerland, 2016b)

ORg:17

G. O' Regan, *Concise Guide to Software Engineering* (Springer, Switzerland, 2017)

Par:01

D. Hoffman, D.L. Parnas, in *Software Fundamentals*, ed. by D. Weiss. Collected Papers by D.L. Parnas (Addison Wesley, Reading, 21)

Par:72

D.L. Parnas, On the criteria to be used in decomposing systems into modules. *Commun. ACM*, **15** (12) (1972)

Par:93

D.L. Parnas, Predicate calculus for software engineering. *IEEE Trans. Softw. Eng.* **19**(9) (1993)

Pol:57

G. Polya, *How to Solve It. A New Aspect of Mathematical Method* (Princeton University Press, Princeton, 1957)

Ros:87

S.M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists* (Wiley Publications, New York, 1987)

Roy:70

W. Royce, *The software lifecycle model (waterfall model)*, in Proceedings of WESTCON (August, 1970)

RuW:10

B. Russell, A.N. Whitehead, *Principia Mathematica* (Cambridge University Press, Cambridge, 1910)

Sha:37

C. Shannon, A symbolic analysis of relay and switching circuits, Masters Thesis, Massachusetts Institute of Technology (1937)

Spi:92

J.M. Spivey, in *The Z Notation. A Reference Manual*. International Series in Computer Science (Prentice Hall, Englewood Cliffs, 1992)

Std:99

Standish Group Research Note, Estimating: Art or Science. Featuring Morotz Cost Expert (1999)

STL:15

Stanford Encyclopedia of Philosophy, Temporal logic. <http://plato.stanford.edu/entries/logic-temporal/>

Tie:91

M. Tierney, The Evolution of Def Stan -55 and -56. An intensification of the formal methods debate in the UK. Research Centre for Social Sciences, University of Edinburgh (1991)

Tur:85

D. Turner, *Miranda*, in Proceedings IFIP Conference, Nancy France, Springer LNCS (201) (September 1985)

Web:93

D. Weber-Wulff, *Selling formal methods to industry*, in FME'93. LNCS 670 (1993)

Wic:00

B.A. Wichmann, A personal view of formal methods. National Physical Laboratory, March 2000

Woo:09

J. Woodcock, P.G. Larsen, J. Bicarregui, J. Fitzgerald, Formal methods: practice and experience. ACM Comput. Surv. **29**

Index

A

Abuse of statistics, 271
Agile development, 12
Algol 60, 234
Alonzo church, 83
Application of functions, 82
Applications of relations, 76
Ariane 5 disaster, 7
Artificial Intelligence, 144
Atelier B tool, 286
Automata theory, 235
Automath system, 260
Axiomatic approach, 46

B

Bags, 159
Bandera, 252
Bijective, 81
Binary relation, 61, 70, 77, 87
Binomial distribution, 270
Booch method, 199
Boole, 95
Boole's symbolic logic, 95
Boyer-Moore theorem prover, 260
B-Toolkit, 286

C

Cadiz, 284
Capability Maturity Model Integration(CMMI),
2, 6, 21, 22, 32, 302
CCS, 54
Central limit theorem, 276
CICS, 43, 164
Class diagrams, 203
Claude Shannon, 97
Cleanroom, 25
Cleanroom methodology, 29
Commuting diagram property, 163
Competence set, 76

Computable function, 83
Computational tree logic, 251
Computer representation of sets, 69
Computer security, 34
Concurrency, 248
Conditional probability, 268
Correlation, 270
Covariance, 269
CSP, 54, 216

D

Darlington nuclear power plant, 43
Data reification, 163
Decomposition, 162
Deduction theorem, 118
Def Stan 00-55, 42
Dependability engineering, 32
Dijkstra, 142, 214

E

Edgar Codd, 76
Endomorphism, 184
Enterprise architect, 287
Equivalence relation, 73
European space agency, 7
Evaluation criteria, 296
Existential quantifier, 105, 122

F

Fagan inspections, 4, 20
Finite-state machines, 55, 236, 237
Flowcharts, 226
Floyd, 225
Formal methods, 22
Formal methods and industry, 292
Formal specification, 39
Frequency table, 277
Functional programming, 82
Functional programming languages, 83

Functions, 78
Fuzzy logic, 134

G

Gaussian distribution, 274
Geometry machine, 258
Gottlob Frege, 101

H

Hackers, 34
Halting problem, 126
Histogram, 277
Hoare, 215
Hoare logic, 227
HOL system, 149, 260
Homomorphism, 184
Hypothesis testing, 278

I

IEEE standards, 9
Indexed structures, 191
Industrial applications of model checking, 252
Industrial applications of VDM, 178
Industrial applications of Z, 164
Industrial tools for UML, 209
Information hiding, 56, 219
Injective, 81
Input assertion, 228
Interactive theorem provers, 259
Interpretation, 125
Intuitionist Logic, 137
Irish School of VDM, 181
Isabelle, 148

L

Laws of probability, 267
L. E. J. Brouwer, 137
Limited domain relation, 76
Linear temporal logic, 250
Logic and AI, 144
Logic of partial functions, 139, 168, 174
Logic programming languages, 145
Logic theorist, 148, 257
Loop invariant., 225

M

Maintenance, 19
Mathematical proof, 47, 164, 255, 262, 301
Miracle, 220
Miranda, 84
Model, 9, 185
Model checking, 56, 57, 245
Model-oriented approach, 46
Models and modelling, 185

Mongolian hordes approach, 1
Monoid, 183

N

Natural deduction, 116
Normal distribution, 274

O

Object constraint language, 208
Object diagram, 203
Object modeling technique, 199
Object-oriented software engineering, 199
Occam's Razor, 186
Output assertion, 228
Overture integrated development environment, 44, 285, 290

P

Paradoxes and fallacies, 91
Parnas, 5, 16, 56, 218
Parnas logic, 141
Partial correctness, 53, 228
Partial function, 79, 156
Performance testing, 18
Pilot, 291
Poisson distribution, 270
Postcondition, 52, 228
Precondition, 52, 53, 228
Predicate, 121
Predicate logic, 105, 121, 300
Predicate transformer, 54
Prince 2, 4, 20
Probability mass function, 269
Probability theory, 266, 280
process calculi, 54
Process maturity models, 21
Professional engineering association, 2
Professional engineers, 6
Project management, 21
Prolog, 146
Proof in propositional calculus, 111
Proof in Z, 164
Propositional logic, 105, 106, 300
Prototyping, 15
Pushdown automata, 239
PVS, 289

Q

Quicksort, 215

R

Random sample, 265, 271, 272, 280
Random variable, 268, 269
Rational software modeler, 287

Rational unified process, 9, 11, 209, 210
Refinement, 40
Refinement in Irish VDM, 193
Refinement in VDM, 177
Reflexive, 71
Reification, 162
Relational database management system, 76
Relations, 70
Requirements validation, 40
Russell's paradox, 68

S

Safety critical system, 33, 36
Schema calculus, 51
Schema composition, 160, 161
Schema inclusion, 160
Schemas, 160
Scientific revolutions, 45
Semantics, 167
Semantic tableaux, 114, 127
Semi-group, 183
Sequence diagram, 205
Sequences, 158
Set Theory, 62
Six sigma, 20
Software availability, 33
Software crisis, 1, 23
Software engineering, 2, 6
Software failures, 7
Software reliability, 25, 27
Software reliability and defects, 27
Software reliability models, 30
Software reuse, 17
Software testing, 17
Specification in VDM, 176
Specifications and proofs, 191
Spin, 252
Spiral model, 10
Sprint planning, 13
Standard deviation, 269, 274
Standish group, 2, 24
State diagrams, 206
Statistical sampling, 272
Statistical usage testing, 29
Statistics, 271
Stoic logic, 93
Story, 13
Strategic defence initiative, 219
Structured query language, 76
Surjective, 81
Syllogistic logic, 90, 91
Symmetric, 71

System availability, 35
System testing, 18

T

Tabular expressions, 218, 230, 234
Tautology, 118
Technology transfer of formal methods, 296
Temporal logic, 135, 250
Test driven development, 18
Theorem provers, 147
Tools for B, 286
Tools for model checking, 288
Tools for theorem provers, 288
Tools for UML, 287
Tools for VDM, 285
Tools for Z, 284
Total correctness, 228
Traceability, 16
Transition function, 237
Transitive, 71
Truth table, 106, 107
Turing award, 216
Turing machine, 241

U

UAT testing, 19
UML activity diagram, 207
UML diagrams, 202
Undefined values, 138
Unified modeling language, 199, 211
Unit testing, 17
Universal quantifier, 105, 122
Use-case diagram, 205

V

Valuation functions, 125
Vannevar bush, 99
Variance, 269, 274
VDM, 40, 49, 167, 179
VDM⁺, 51, 182
VDM tools, 285
Vienna development method, 49, 59, 167, 168, 301
VIPER, 47

W

Waterfall model, 9
Weakest precondition, 53
Weakest precondition calculus, 218

Y

Y2K, 3, 7
Y2K bug, 7

Z

Z, [40](#)

Zermelo set theory, [52](#)

Z/EVES, [285](#)

Z formal specification language,
[151](#), [164](#)

Z specification, [51](#)

Z specification language, [51](#), [216](#)