Andrzej Wąsowski
Thorsten Berger

Domain-Specific Languages

Effective Modeling, Automation, and Reuse

Springer

Domain-Specific Languages

Andrzej Wąsowski • Thorsten Berger

Domain-Specific Languages

Effective Modeling, Automation, and Reuse

Springer

Andrzej Wąsowski
Department of Computer Science
IT University of Copenhagen
Copenhagen, Denmark

Thorsten Berger
Faculty of Computer Science
Ruhr University Bochum
Bochum, Germany

ISBN 978-3-031-23668-6 ISBN 978-3-031-23669-3 (eBook)
https://doi.org/10.1007/978-3-031-23669-3

Cover illustration: The cover design is based on work done by Sonja Israel, Ruhr University Bochum, for the Authors, © The Authors, 2022, all rights reserved.

The wiring diagram of Porsche 911E in Fig. 1.1 is included by permission of Corporate Archives Porsche AG. | The photograph of Porsche 911 in Fig. 1.1 is courtesy of Wikipedia User Thesupermat under CC BY-SA-3.0 license https://en.wikipedia.org/wiki/Porsche_911#/media/File:Bonhams_-_The_Paris_Sale_2012_-_Porsche_912_‘SWB’_Coup%C3%A9_-_1967_-_014.jpg. | The image of Charles Babbage on p. 6 is a public domain image, obtained thanks to the Wikimedia Foundation (https://en.wikipedia.org/wiki/Charles_Babbage#/media/File:Charles_Babbage_-_1860.jpg). | The dance choreography diagram of Fig. 1.6 is by Wikimedia Commons contributor Inigolv, released under CC BY-SA 4.0 license, https://commons.wikimedia.org/wiki/File:Labanotation.svg. | The photo of Noam Chomsky on p. 97 is licensed as CC BY-SA 4.0 by Wikipedia user Σ. | The program in Fig. 10.23 on p. 380 comes from Schoeberl [1], reproduced with permission. | The image in Fig. 11.1 is courtesy of Ştefan Stănciulescu, used with permission. | All other images and photographs were created by the authors.

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

Foreword

A Shiny New DSL Textbook in Town

Advanced textbooks are crucial in maturing fields in computer science. In this sense, Wasowski and Berger's book makes a significant contribution to maturing the notion of DSL[1] in a contemporary way; the book also sends SLE[2] on an impressive, cross-technological space mission.

A younger Ralf was slightly obsessed with PL[3] textbooks and I found each of those books back then to be important in complementary ways. For example, I was fond of Carl A. A. Gunter's 1992 book on semantics due to its elegant mathematical approach. I loved Peter Mosses' 2005 book on action semantics for its clever "ontology" of language concepts. I also discovered Benjamin C. Pierce's 2002 TAPL book with great relief, as it was moving the meta-theoretical realm from art to science or engineering. The DSL book at hand is, of course, more of an SE[4] than a PL book, but it's a bit of a hybrid because it's a true SLE book.

Everyone should rest assured that we need several books to cover the field of SLE and, more specifically, the field of DSLs. The DSL topic is covered somewhat by scholarly articles, surveys, and PhD theses. There isn't enough available in textbook form. Anneke Kleppe's 2008 book "Software Language Engineering: Creating Domain-Specific Languages Using Metamodels" was a good starting point. Compared with the DSL book at hand, Kleppe's book is outdated, highly technological space-specific (i.e., one of variant of MDE[5]), and incomplete. Kleppe's book lacks, for example, coverage of foundations, detailed guidelines for QA[6], exercises, and further reading pointers, when compared to Wasowski and Berger's book at hand. Markus Völter et al.'s 2013 book "DSL Engineering: Designing, Implementing and Using Domain-Specific Languages" is more up-to-date and more comprehensive, but it's also driven by specific technologies and it does not serve the classroom.

There is also my own 2018 textbook "Software Languages: Syntax, Semantics, and Metaprogramming" (for short "the SL book") which is somewhat close to Wasowski and Berger's book at hand—both from a community perspective and in terms of structure and contents, but there are powerful differences. A crucial advancement of Wasowski and Berger's book is that it presents a modern notion of

[1] DSL: Domain Specific Language
[2] SLE: Software Language Engineering
[3] PL: Programming Language
[4] SE: Software Engineering
[5] MDE = MD(S)E: Model Driven (Software) Engineering
[6] QA: Quality Assurance

DSL *comprehensively* from an *SE* point of view; it also realizes an important component of the SLE mission: *bring together the distinct technological spaces PL and MDE* (grammarware versus modelware or algebraic data types versus meta-models).

I wholeheartedly welcome this new book and cherish its original qualities; see the detailed discussion below.

Structure of the DSL Book

Let me start with a brief recap of the DSL book's structure.

- Chapters 1 and 2 **introduce to and motivate the DSL subject**. The first chapter ties up DSLs with MDE and the second chapter summarizes the life-cycle of realizing modeling languages / DSLs—with the "modeling languages hat" on.
- Chapter 3, perhaps unsurprisingly, kicks off the DSL development life-cycle with "**abstract syntax**." In addition to the obvious formalistic aspects, considerable emphasis is placed on "domain analysis." For example: What are the concepts that the DSL needs to address? What I'd like to call SLE-style "technological space liberation" starts no later than here. That is, it's great to see how meta-modeling (MDE) and algebraic data types (PL) are demonstrated as alternative paths towards (abstract) syntax modeling and implementation.
- Chapter 4, quite logically, addresses "**concrete syntax**." Just like the previous chapter, the meta-meta-level is covered. (Thanks!) The chapter also gets quite serious on grammar engineering. For instance, it covers syntax-oriented testing and left-recursion removal. (I love it!) Two distinguished paths—the one via a parser generator (in the MDE setting) and the other via a parser-combinator library (in the PL setting)—are exercised. In summary, both the PL and MDE point of view regarding concrete syntax are served back to back. (I submit: To count as a SLE/DSL engineer in 2022, you certainly need to handle abstract and concrete syntax across technological spaces.)
- Chapters 5 and 6 cover "**static semantics**" while managing the identification of two layers: one with involvement of type systems higher up (and more intricate) and one with just more basic constraints (à la "well-formedness") as a starting point. Formalistically, static semantics is presented as a logical constraint problem where PL and MDE realizations are demonstrated. For instance, Scala can encode constraints on algebraic data types and UML's OCL can express constraints on meta-models.
- Chapters 7 to 9 cover "**dynamic semantics**." More specifically, Chapters 7 (Transformation), 8 (Interpretation), and 9 (Generation) develop different paradigms for the design and the implementation of dynamic semantics while also covering testing and other types of QA for such components. Again, both PL (e.g., Scala) and MDE (e.g., ATL) are exercised. Chapter 10 focuses on an important PL-specific class of DSL implementation options—**internal DSLs**—that leverage meta-programming; both the deep and shallow options are covered.
- The book exercises a number of domains, for example, robotics, but Chapters 11 to 13 engage with a distinguished domain: "**variability**" in software development, which is indeed a (DSL!) domain in so far that variability approaches tend to use DSLs for configuration. As a bonus, the book rehashes the central underlying notions of "software product line" and "feature modeling." In the last chapter, the book lifts variability to the language level, thereby arranging DSLs in "linguistic product lines". (This is great!)

Original Qualities of the DSL Book

- The DSL book shows a rich PL approach (i.e., Scala with algebraic data types + libraries) and the de-facto standard MDE approach (i.e., EMF and friends) back to back. (Admittedly, my SL book is a bit more of a PL book.)
- The DSL/SLE/MDE fields are quite rich regarding technologies and formalisms ("notations") so that a textbook could easily drown in formalistic matters and idiosyncrasies. However, the authors do a great job abstracting from the technologies chosen for illustration; the authors also mix in coverage of soft parts, for example, requirements or domain analysis or the "bigger picture" in terms of meta-meta-level considerations and multi-level modeling views.
- The authors spell out QA guidelines and methods for each and every DSL component designed and implemented. This coverage goes beyond simple unit testing (e.g., for transformations); it covers a range of properties, for example, robustness, well-formedness, completeness, and correctness. (Admittedly, my SL book does not discuss QA so broadly.)
- In fact, the book is generally engineering-oriented (i.e., SE-oriented) in that it systematically exercises problem-oriented thinking of software engineers while it drives the reader towards mapping problems to "standardized" solutions within the PL and MDE spaces. As mentioned before, the solution spaces chosen—Scala versus EMF—are discussed at a good level of abstraction. Thus, the reader can imagine alternative solutions. (For instance, things make sense for me as a rusty Haskell programmer.)
- On top of strong coverage of the topics of transformation and interpretation, the authors develop the generation topic in an exceptionally strong manner. In particular, they explain systematically, how and why (!) to mix templates and recursion and how to reuse an existing interpreter for arriving at a generator. (This is very insightful!) Further, the authors don't fall into the trap of engaging into a compiler construction-like optimization discussion.
- The book makes good use of example code and case studies. It uses simple toy examples to develop the basic skills. It scales up to more significant case studies to cover different DSL domains and to prove the scalability of the methodology. The running robotics theme is definitely fun for most readers. Ultimately, the book engages in the "variability" domain; see above. All example code from the book is available for download!
- The book is clearly classroom- and learning-oriented in that the writing style is skill-driven and there are loads of exercises that directly connect to the developed examples. This amazing level of exercise support reminds me of textbooks in mathematics. (Solutions to the exercises aren't (yet) available online.) The book is also immediately fit for research-oriented ($\geq$ M.Sc. level) courses in that it provides references to underlying research foundations and further-reading material.

Reading Never Stops

The DSL book is a perfect testament of how the MDE, PL, and SE communities have grown together at the pivotal front of SLE. The authors have a very strong standing in the SE and MDE fields with significant engagements border-lining to PL. The SLE community, "by design" (see [1]), aims at such community integration for the best of software languages (DSLs very much included!) on an engineering front. The authors' further reading pointers, per chapter, serve well to maintain and improve the further integration of the fields both in terms of foundations and engineering.

My own background differs a bit from the one of the authors in that I am more of a PL, grammarware, compiler and program transformation person. Also, more recently, I have turned into a "megamodeling" aficionado. Accordingly, I would like to submit a few further reading pointers that, in my view, logically continue the directions given in the book. Please accept my apologies for the use of self-references; the assumption is here that the self-referenced papers discuss or reference also fundamental work by others.

- Model generation pops up a number of times in the DSL book in the context of testing-based QA for robustness or correctness, for example, when testing model transformations. More of the results of grammar-based test-data generation [7] (e.g., regarding coverage criteria, use cases regarding code generation, and coverage of semantic constraints) can be incorporated into the DSL engineering methodology. Also, generation of graph-shaped data, which the book understandably identifies as more complex than generation of tree-shaped data, has been researched in the (OO) programming context [9].
- Much of the variability-inspired reuse discussion in Chapters 11 to 13 is focused on and tailored to models (e.g., how to support variability in models), while reuse of semantics is largely covered in the sense of "interpreters and code generators are programs, too" and can be thus turned into software product lines. There is, however, a great history of reuse (modularity, de-/composition) for language definitions and, specifically, semantics descriptions, for example, action semantics [11], Modular SOS [10], transformational approaches [4, 3], and paradigm shifts for attribute grammars [6, 8]. One might argue that the PL field (including language definition and implementation) has had feature models and product lines for language definitions for a long time.
- The DSL book exercises the notions of "meta-modeling hierarchy" and "language-conformance hierarchy" to capture the linguistic architecture of DSL solutions. (I very much appreciate this engagement!) This type of discussion is an instance of the more general paradigm of "megamodeling" or "linguistic architecture" [2, 5], which by its further development and deployment could help in formalizing DSL methodology in an executable manner. For instance, we could explain bootstrapping more rigorously by means of megamodeling, when compared to what's done in the textbook at hand.

Final Verdict

I concur with the greatest German comedian, Otto Waalkes, who says (I translate): "There is a trend towards a second book."[7] Assuming that my SL book is already in your library, you will definitely want to add the DSL book by Wasowski and Berger. All jokes aside, the new DSL book by Wasowski and Berger is most definitely a great textbook covering the DSL topic—rather than the less clearly demarcated

[7] In German: "Der Trend geht zum Zweitbuch." It appears that the phrase expresses fear of the opinions of the illiterate man who has only read a single book. Historically, that single book may have been "the" bible.

SL topic—in a profound manner, while also conveying importance of the DSL topic in today's software development practice. This book excels in delivering state-of-the-art SE research to teaching (learning) and SE practice. Finally, this DSL book is a great SLE book, too.

Ralf Lämmel
Professor of Computer Science (Software Languages)
University of Koblenz
laemmel@uni-koblenz.de
Author of the Software Languages Book, http://www.softlang.org/book

References

[1] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter. "Guest editors' introduction to the special section on software language engineering". In: *IEEE Trans. Soft. Eng.* 35.6 (2009), pp. 737–741 (cit. p. vii).

[2] Frédéric Jouault, Bert Vanhooff, Hugo Brunelière, Guillaume Doux, Yolande Berbers, and Jean Bézivin. "Inter-dsl coordination support by combining megamodeling and model weaving". In: *ACM Symposium on Applied Computing (SAC)*. 2010 (cit. p. viii).

[3] Ralf Lämmel. "Declarative aspect-oriented programming". In: *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. Technical report BRICS-NS-99-1*. University of Aarhus, 1999, pp. 131–146 (cit. p. viii).

[4] Ralf Lämmel. "Evolution of rule-based programs". In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 141–193 (cit. p. viii).

[5] Ralf Lämmel. "Relationship maintenance in software language repositories". In: *Art Sci. Eng. Program.* 1.1 (2017), p. 4 (cit. p. viii).

[6] Ralf Lämmel and Günter Riedewald. "Reconstruction of paradigm shifts". In: *2nd Int. Workshop on Attribute Grammars and their Applications, WAGA'99*. 1999 (cit. p. viii).

[7] Ralf Lämmel and Wolfram Schulte. "Controllable combinatorial coverage in grammar-based testing". In: *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006*. Vol. 3964. Lecture Notes in Computer Science. Springer, 2006 (cit. p. viii).

[8] Johannes Mey et al. "Relational reference attribute grammars: improving continuous model validation". In: *J. Comput. Lang.* 57 (2020), p. 100940 (cit. p. viii).

[9] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. "Korat: A tool for generating structurally complex test inputs". In: *29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 2007 (cit. p. viii).

[10] Peter D. Mosses. "Modular structural operational semantics". In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 195–228 (cit. p. viii).

[11] Peter D. Mosses. "Theory and practice of action semantics". In: *Mathematical Foundations of Computer Science, MFCS'96*. Vol. 1113. Lecture Notes in Computer Science. Springer, 1996 (cit. p. viii).

Preface

What is a domain-specific language (DSL)? How do we use modeling for building systems effectively? How are domain-specific languages relevant in practice? How complex and expensive is developing languages? Does model-driven software engineering (MDSE) impose a waterfall process with much overhead? Can it be iterative, agile, test-driven? This book attempts to answer these and many other questions in a format that is suitable for academic teaching and learning. We have designed it around three principles: sound pedagogy, grounding in research, and being non-sectarian regarding programming paradigms.

Pedagogy. We develop a skill-oriented, operational, and experiential approach rooted in Scandinavian educational tradition. We do not want to *tell you about* DSLs and MDSE. We want *you* to *practice building DSLs*, at your keyboard, in your problem domain, using your favorite programming language. We make you manipulate and analyze software artifacts automatically, including validation of properties, modifying models or data, computing new artifacts out of existing ones, or generating code. We emphasize practical take-home lessons, including toolbox-like design guidelines, often derived from empirical research by others.

To facilitate constructive learning, we try to avoid presenting concepts without examples and exercises. We developed dozens of concrete and reproducible examples, fitting each into a small figure, presenting key concepts without excessive noise. This was a challenge in itself, as language projects tend to be too large to present concisely. Still, our examples use concrete modern technology. We focus on tools that are accessible for professional use, stable, and not controversial to adopt. We prioritize language-engineering features available in mainstream programming languages over academic solutions closer to our daily research work. We have also designed 277 exercises, most focusing on training a single skill. We believe that establishing fine-grained exercises is a stepping stone in developing a pedagogy of any technical field. We hope that the community will recognize this contribution.

To the best of our knowledge, this is the first book that systematically discusses practices of testing language designs and model-driven systems. This content is distributed throughout all the chapters; quality assurance in a software project is not an add-on, which could be delegated to a dedicated chapter as an afterthought, but something we need to practice throughout the project. The book's exercise collection also incorporates tasks on testing and other quality assurance practices.

Research foundation. To prevent the skill-orientation from hampering the durability of the contents, we counterbalance it with a focus on fundamental concepts and principles. We ground the text in programming-language and model-driven-engineering research, avoiding buzzwords and acronyms. Each chapter concludes with an overview of the related literature, commenting on key papers and books. It directs students and early-stage researchers working on theses and research projects to the relevant literature of the field. If you seek a book that will quickly teach you to use a particular library, framework, or tool, then stop reading now. We want you to gain knowledge that will survive changes of technology, current programming languages and tools; that will help you critically assess different tools and strategies; indeed, that will allow you to build the next generation of languages and tools.

Programming paradigms. Domain-specific languages are studied in several overlapping research communities: object-oriented programming, functional programming, dynamic languages (Racket, Smalltalk), modeling, and software-language engineering. We present key perspectives from multiple fields, unifying terminology and setup. We discuss both internal and external DSLs, using both object-oriented and functional programming languages, exploiting both model-based and grammar-based technology. We use standard compiler construction terminology when appropriate, unifying it with vernaculars of the separate communities (for instance, relating meta-model, instance, abstract syntax, or relating meta-programming with transformation). In the interest of trustworthiness and durability, the terminology is cleaned off acronyms and buzzwords as much as possible.

Teaching Guide

Target audience and prerequisites. This book has been developed in a series of courses on MDSE and DSLs at the IT University of Copenhagen in Denmark, and at Chalmers University of Technology and the University of Gothenburg in Sweden. We aim specifically at senior undergraduate and junior graduate students in Computer Science or Software Engineering. We have included examples and lessons from industrial and open-source projects, and from industrial research—thus we hope that practitioners will also find this a useful reference. Numerous exercises and the associated code repository facilitate self-study, as well.

The text is designed for a reader with basic experience of object-oriented and functional programming, including testing. No significant mathematical background is expected, beyond what is typically a prerequisite for an algorithms course: propositional logic, quantifiers, sets, functions, relations, and basic data structures.

Teaching with this book. We used this material to execute courses lasting 14-15 weeks, of which 60-70% comprised lectures and labs based on the book and the rest were reserved for a larger project. The entire book supports a course on external and internal DSLs, in object-oriented and functional style, complemented with product-line engineering as a simple but effective case of a DSL-based architecture. However, depending on the school, the program, and the teacher's preferences, different smaller subsets can be selected:

- *A classical DSL course:* Chapters 1–10. In this variant, about 7-10 weeks of classes are needed. Chapter 6 on typing is more advanced. As many useful DSLs can be developed without type checking, it can be skipped. Similarly, Chapter 10 can be dropped, if internal DSLs are not of interest. The intensity can also be lowered by omitting the functional programming examples and sections.
- *A focused introduction to internal DSLs:* Chapters 1–4, (optionally 5–6), and 10.
- *A focused course on product-line engineering:* Chapters 1–3 and 11–13.

In our experience, the lectures and exercises do not sufficiently facilitate learning in a DSL course. Thus we normally supplement them with:

- *Quizzes and discussion sessions in class*, often during lectures. This book includes many exercises, and many of them were actually developed for in-class use. We used very short pen-and-pencil, closed-answer, or discussion exercises in class.

- *Recap lectures and technical briefings.* A few days after the main class, a short recap class summarizing the past week's material, followed by a briefing where the students can ask questions about the tooling.
- *Weekly assignments.* Exercises from the book are selected for homework, to make students actively experience the material. We include many exercises, to let the teacher choose suitable ones for her audience. The exercises use different languages, or allow a programming language to be selected. It is also easy to re-target most exercises to another language. Obviously, students should not be made to use all the programming languages of the book, but rather the main language to work with should be chosen to make sense for their context.
- *A language engineering project.* Small exercises facilitate learning individual skills but may obscure the big picture. This is best addressed by an end-to-end project, typically appreciated by students. In the project, they design and build a DSL for a concrete problem, and obtain implementations for a few input models.
- *Exams:* We have used both written exams (many of the exercises in the book have been developed for past exams), and oral exams that were based on defending the project work and answering basic questions about the material.

Online material. The book website can be found at http://dsl.design/. We will be releasing additional material there: slides, extra chapters, and links to open supplementary material created by others. Our code repository is found at https://bitbucket.org/dsldesign/dsldesign/src/. It contains build infrastructure and source code for most of the book's figures, examples, and exercises. The repository is organized using `gradle` as a build system, but incorporates code in many languages. Notably, the repository does not require working with Eclipse or any other particular IDE. Consult the `README` file for details.

We link to the repository from figures and text (clickable links). For brevity we omit the prefix "`https://bitbucket.org/dsldesign/dsldesign/src/master/dsldesign.`" For example, prpro/model/prpro.ecore links to https://bitbucket.org/dsldesign/dsldesign/src/master/dsldesign.prpro/model/prpro.ecore. The code is released under the Apache 2.0 license. We receive contributions and comments gladly.

Acknowledgements

We thank all students and colleagues who helped to create this book by finding errors, contributing examples, and participating in stimulating discussions. We owe you: Abdulrashid Mas'ab Mohammed, Alexandru Florin Iosif-Lazar, Anders Fischer Nielsen, Cem Turan, Christoffer Stougaard Pedersen, Daniel Strüber, Erik Meijer, Francisco Martínez Lasaca, Georg Hinkel, Hjalte Sorgenfrei Mac Dalland, Jalil Boudjadar, Jean Bezivin, Jean Privat, Jeremy Gibbons, Jonatan Gustafsson, Jurgen Vinju, Karol Wąsowski, Kasper Hansen, Marek Furák, Martin Schoeberl, Miëtek Bak, Oscar Jönsson, Peter Sestoft, Ralf Gerstner, Ralf Lämmel, Robert Palm, Robin Bellini Olsson, Rolf-Helge Pfeiffer, Ştefan Stănciulescu, Sven Peldszus, Tijs van der Storm, Titus Barik, Tobias Schwarz, Vadim Zaytsev, and typeswitch (of Twitter). Several hundred students in Copenhagen and Gothenburg have bravely taken our courses, being the main motivation for us, while also serving as beta-testers. Phil Watson, our copy editor at Springer, not only mercilessly pointed out wrong articles, misspellings, and hyphenation errors, but also identified problems in arguments and formulae. We thank you all. All remaining flaws are ours.

Andrzej Wąsowski, Thorsten Berger

Copenhagen, Bochum, September 2022

Contents

To Aleksandra, Urszula, Jakub, and Karol
Thank you for all the support and understanding.
Andrzej

To Maria, Mira, Ingeborg, and Günter
Thank you for all the understanding and support.
Thorsten

Language is sufficient to any thought.
Imperfect expression is the fault of limited writers,
not limited language.
(Francis-Noël Thomas and Mark Turner)

I wish to approach truth as closely as possible,
and therefore I abstract everything until I arrive
at the fundamental quality of objects.
Piet Mondrian

1 Using Modeling Languages

Using models to design complex systems is common in many engineering disciplines, including architecture (buildings), civil engineering (roads and bridges), automotive engineering (cars), and avionics (airplanes). Models have an ever-growing list of applications. Engineers build them to assess system properties before prototyping or to steer construction, production, and servicing processes. For one system, different kinds of models may be built, each providing a different perspective. For instance, three-dimensional models are used when designing the chassis of a car, while analog-circuit models describe its electrical system. Blueprint models are used in production, while yet different ones, such as maintenance and service models, are used later when servicing systems. All these examples of models describe structural and functional properties of real-world systems. However, models can also be used to describe and assess rather intangible properties that are neither structural nor functional, such as system reliability, power consumption, efficiency, or production cost. We say that models are *purpose-specific* and *domain-specific*: they are tailored for a given purpose and represent the main characteristic aspects of the domain. For example, telephone-network switching models are different from railway-track switching models.

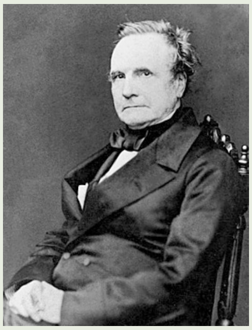
Definition 1.1. *A* model *is an abstraction of reality made with a given purpose in mind.*

The main purpose of using models is to combat complexity: of the problem, of the solution design, and of the system implementation or production. The understanding of complex problems, solutions, designs, and processes is possible thanks to *abstraction* [56]. Abstraction is the simplification and elimination of information with respect to a given purpose. A model does not contain all information, but it preserves the information necessary to perform the intended application. We can say that *"all models are wrong, but some are useful"* [7]. For instance, aesthetic information is typically not necessary to assess performance.

Models can not only hide information, they can also *approximate*. For instance, the Newtonian gravity model is sufficiently precise for applications in mechanical engineering. It is widely applied, although we know that it is imprecise. It is crucial that both abstraction and approximation are not adverse to the purpose of a given model. Abstraction should not hide relevant information, and approximation should only lead to acceptably small errors.

Thanks to the rapid growth of computing technology, models are increasingly often digital. In fact, most engineering models are digital today,

Model-Driven Engineering Prehistory


Source: Wikimedia Foundation

Charles Babbage (1791–1871) was an English mathematician, philosopher, and mechanical engineer, credited with designing (not building) one of the earliest examples of a mechanical computer, a *difference engine* or a machine to automatically compute numeric tables of mathematical functions using polynomial approximations [1].

Interestingly, Babbage's reasons to build the difference engine resembled the motivation of most automation projects, including model-driven software engineering. In the nineteenth century, mathematicians would calculate approximations of irrational functions manually. The results of these calculations, multi-page tables of numbers, would then be typeset by printers and printed on paper, so that engineers could use them in calculations. Babbage found this tedious and error prone—an ideal candidate for automation.

His machine worked as follows. First, it would calculate the values for the tables, and a custom-built printing back-end would typeset results correctly, so no errors could sneak in at that stage. He even considered what print colour should be used to minimize the number of errors. Today, when we search for MDSE opportunities, we seek software development activities that are tedious and error prone, like creating lots of boilerplate code. We use automation not only to derive early designs, but for end-to-end construction of effectively functioning systems.

The British government showed interest in Babbage's project, believing that this could bring down the cost of computing the numeric tables. Modern managers use the same reasoning when considering the introduction of MDSE. Unfortunately, Babbage had not managed to realize his detailed designs in practice, so he did not know whether the benefits were actual. For MDSE, we fortunately know that there are substantial gains in quality and cost to be reaped. We briefly survey them in Sect. 1.2. (Babbage's difference engine has been built twice in modern times, following his blueprints. One can appreciate it in the Science Museum in London and in the Computer History Museum in Mountain View, California.)

even if they describe physical artifacts, such as buildings or car engines. Building computerized models is cheaper than building physical models. It allows animation, simulation, and computation of non-structural and non-functional properties, for instance weight.

While computerized models have taken over other engineering disciplines, their use in software engineering is also growing. Many software engineers use purposeful abstractions of design and computation without thinking of them as models. For instance, relational-database queries are models, and so are HTML web pages and their style sheets. Reactive algorithms, or behavior of software in general, are often described using automata models. Efficiency of algorithms is approximated using asymptotic complexity models. In this book, we shall look at multiple opportunities to use models in computing and to introduce purposeful domain-specific modeling languages (DSMLs or DSLs for short) into the development of software systems.

Software engineers face the same complexity problems as engineers in other disciplines do. In many ways, software systems are just as complex—often even more complex—than other achievements of engineering. Many

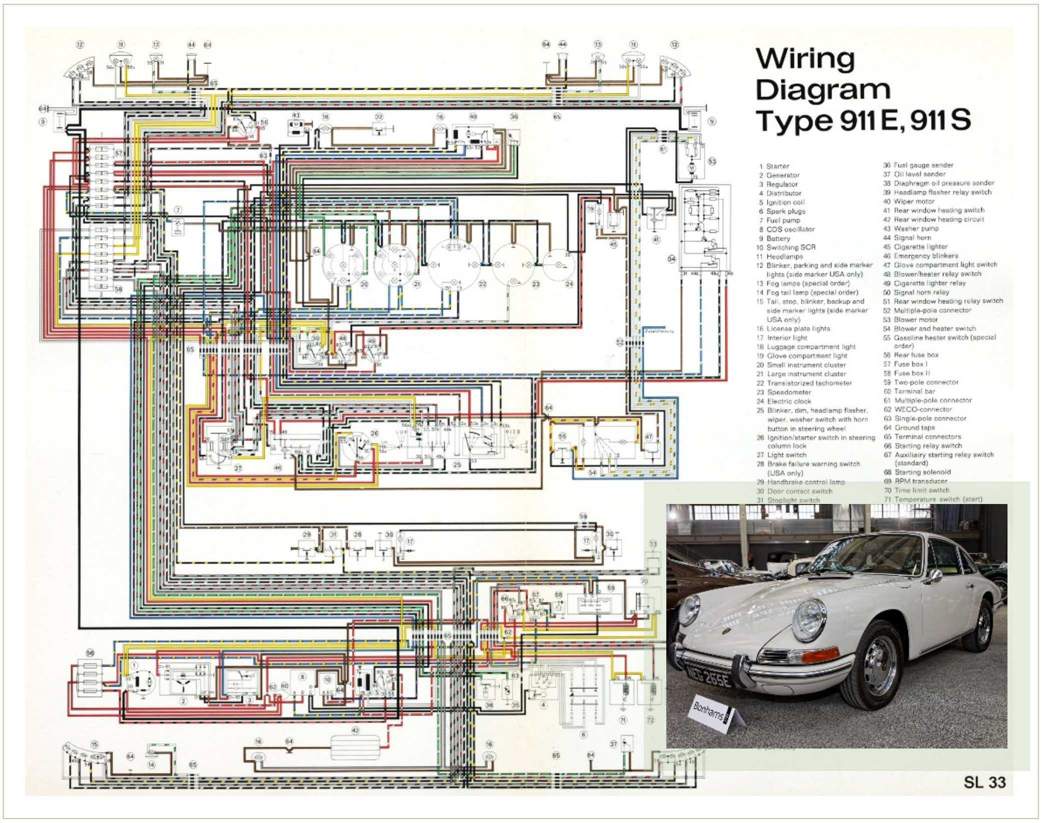

Wiring Diagram Type 911 E, 911 S
1 Starter
2 Generator
3 Regulator
4 Distributor
5 Ignition coil
6 Spark plugs
7 Fuel pump
8 CDS oscillator
9 Battery
10 Switching SCR
11 Headlamps
12 Blinker, parking and side marker lights (side marker USA only)
13 Fog lamps (special order)
14 Fog tail lamp (special order)
15 Tail, stop, blinker, backup and side marker lights (side marker USA only)
16 License plate lights
17 Interior light
18 Luggage compartment light
19 Glove compartment light
20 Small instrument cluster
21 Large instrument cluster
22 Transistorized tachometer
23 Speedometer
24 Electric clock
25 Blinker, dim, headlamp flasher, wiper, washer switch with horn button in steering wheel
26 Ignition/starter switch in steering column lock
27 Light switch
28 Brake failure warning switch (USA only)
29 Handbrake control lamp
30 Door contact switch
31 Stoplight switch
36 Fuel gauge sender
37 Oil level sender
38 Diaphragm oil pressure sender
39 Headlamp flasher relay switch
40 Wiper motor
41 Rear window heating switch
42 Rear window heating circuit
43 Washer pump
44 Signal horn
45 Cigarette lighter
46 Emergency blinkers
47 Glove compartment light switch
48 Blower/heater relay switch
49 Cigarette lighter relay
50 Signal horn relay
51 Rear window heating relay switch
52 Multiple-pole connector
53 Blower motor
54 Blower and heater switch
55 Gasoline heater switch (special order)
56 Rear fuse box
57 Fuse box I
58 Fuse box II
59 Two-pole connector
60 Terminal bar
61 Multiple-pole connector
62 WECO-connector
63 Single-pole connector
64 Ground taps
65 Terminal connectors
66 Starting relay switch
67 Auxiliairy starting relay switch (standard)
68 Starting solenoid
69 RPM transducer
70 Time limit switch
71 Temperature switch (start)
SL 33
NEB 255E
Bonhams

Figure 1.1: An early example of a model, the electrical scheme of the Porsche 911E, contrasted with the actual system. Note the deliberate loss of information between the model and the real system, and how the model supports a well-defined purpose. Source of wiring diagram: Corporate Archives Porsche AG. Source of Porsche photo: Wikipedia User Thesupermat (CC BY-SA-3.0).

commercial software systems have more lines of code than a Boeing 747 has mechanical parts. In fact, the Jumbo Jet has 'only' six million parts, half of them being super-simple fasteners, many of them identical. In contrast, the Linux kernel had about 15 million lines of code by December 2014. This complexity is (partly) controlled using a configuration model and an automated build process [3]. The Open Office productivity package had reached nine million lines of code in 2012. Microsoft Windows is reported to have exceeded 50 million lines in 2003. *"Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques"* [56], in order to combat the complexity.

1.1 Model-Driven Software Engineering

Software development is particularly well aligned with modeling. For a car, there is a notable abstraction gap between a model and a real physical construction (Fig. 1.1). For software, both models and systems are digital—

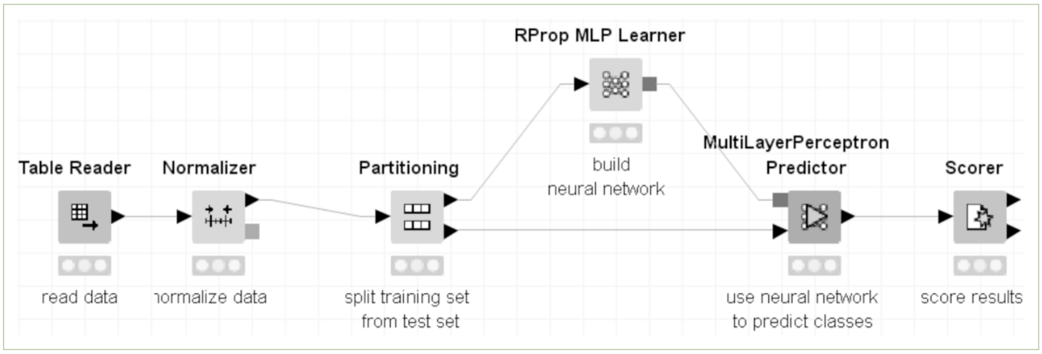


Figure 1.2: *KNIME facilitates data analytics and AI development with a graphical DSL*

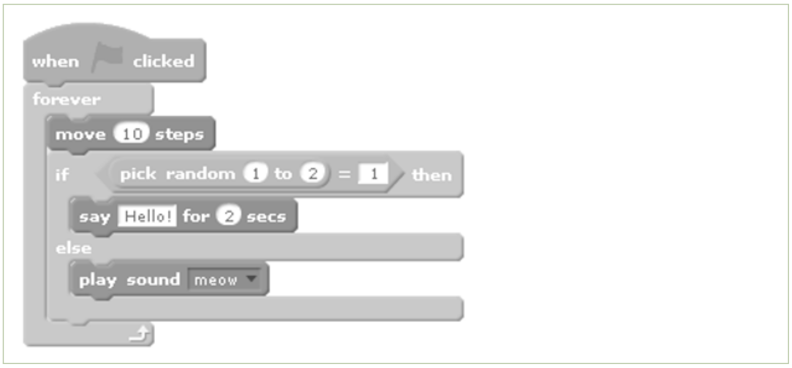
the gap is much smaller; everything is a model in software engineering [4]. One can refine a model into a system following a stepwise and automatic manner, and with much less effort than when designing cars or buildings. After all, source code is also a model that abstracts many aspects of a physical computation, just containing enough details to run the computation. This proximity of models and programs allows us to make modeling the central paradigm in development: the Model-Driven Software Engineering.

Definition 1.2. Model-Driven Software Engineering (MDSE) *is a method to produce software by creating and exploiting models. The focus is on models, modeling, and model analysis, as opposed to programs and programming. MDSE automates code production and other development activities.*[1]

Organizations engineering software can adopt MDSE either by relying on off-the-shelf modeling techniques and languages that have been developed by others, or by creating their own languages and modeling infrastructure, thanks to powerful MDSE tools and techniques, which we will present in this book. As confirmed in surveys in industry [31, 39, 65], the most well-known off-the-shelf modeling language is the Unified Modeling Language (UML) [48, 20, 47], established and developed under the umbrella of the Object Management Group (OMG). UML includes over 13 different languages, such as class diagrams or sequence diagrams, which developers can use without having to design languages and build tools, such as editors, to apply MDSE. In many cases, however, it is desirable to create languages specifically tailored to one's need, such as a language that customers can use to configure a product. Let us now look at some existing languages to get a better intuition about their look and feel.

Example 1. KNIME[2] is an extensible platform for composing data-analytics pipelines, used by data scientists for building data analyses, visualizations, classifications, preprocessors, and other tasks in pharmaceutical, business intelli-

[1] Inspired by definitions of Selić [56] and of Wikipedia editors (https://en.wikipedia.org/wiki/Model-driven_engineering, retrieved 2022/08).



```
when clicked
forever
  move 10 steps
  if pick random 1 to 2 = 1 then
    say Hello! for 2 secs
  else
    play sound meow
```

Figure 1.3: Scratch facilitates end-user development with a graphical DSL

gence, and financial research. Figure 1.2 shows an example model of an image classifier in the DSL of KNIME. Nodes represent data-processing functionalities and edges the data flows. The first node, Table Reader, obtains a data set, which is then normalized by a Normalizer, partitioned into training and test sets (1/3 vs 2/3, held in the node properties not shown in the figure), and fed into the MLP Learner, a multi-layer perceptron. Subsequently, the Predictor uses the trained perceptron to classify test images, while the Scorer assesses the quality of the results, calculating accuracy statistics (e.g., recall, precision, F-measure). Given a dedicated language, like KNIME, inexperienced programmers can focus on the problem domain as opposed to low-level architectural interfaces.

Example 2. Scratch[3] is another end-user-oriented modeling language. Unlike KNIME, Scratch is an imperative language, composed of control blocks known from programming languages. Scratch programs resemble jigsaw puzzles (Fig. 1.3). The syntax is designed to meet the expectations of the target user group, the school children. Using a domain-specific syntax matching the user's expectations is one of the key success factors in designing DSLs. Scratch boasted over a million users in 2014. The models are hosted on the Scratch website and are freely accessible to the public. While Scratch is Turing-complete, it is still not a GPL. It is not supposed to be used to write general programs, but focuses on game-like sprite programs written for learning.

Example 3. Google Protocol Buffers are a data-modeling language aimed at flexible and efficient serialization and persistence of structured data across multiple programming languages and platforms. An example is shown in Fig. 1.4. Since the language was initially developed for passing messages between different machines in a request/response protocol, it uses 'message' as a metaphor for a data structure. It is a textual language (syntax is expressed as a stream of characters), an *interface definition language*, and a *structural modeling language* (a competitor of, for instance, XML). However, protocol

[2] https://www.knime.com, retrieved 2022/08
[3] http://scratch.mit.edu, retrieved 2022/08

```
message Person {

    enum PhType { MOBILE = 0; WORK = 1; }

    message PhoneNo {
        required string no = 1;
        optional PhType type = 2 [default = MOBILE];
    }

    required string name = 1;
    repeated PhoneNo phone = 2;
}
```

Figure 1.4: *An example model in the Google Protocol Buffers language*

```
class Client < ActiveRecord::Base
    has_one  :address
    has_many :orders
    has_and_belongs_to_many :roles
end
```

Figure 1.5: *A simple data model in Ruby on Rails, using an internal DSL embedded in Ruby syntax*

buffers are small and clean, use very little bandwidth for transmitting the data, and have a human-readable syntax for their schema, unlike XML. The example model describes a `Person` structure with the two properties `name` and `phone`, where the latter is a message itself (a substructure) of type `PhoneNo`. The data elements described by this model are assigned ordinal numbers representing their placement in the serialization, which allows fields to be reordered and renamed without changing the message format. The format also allows optional elements to be assigned default values. It is an important design criterion for the message serialization domain to allow as much backwards compatibility as possible, when the message format changes or its definition is refactored. Implementing a proper message format serialization infrastructure with these properties is actually cumbersome, even if it is needed in many software projects. Protocol buffers have a standalone implementation for at least Python, Java, and C++. They demonstrate an important motivation for DSLs and modeling: *code reuse*. The protocol buffers libraries have been implemented once and for all and are maintained in only one place, saving a lot of effort duplication. Since they are used by many, the libraries are of considerably higher quality than if they were reimplemented repeatedly in different projects. In February 2014, there were 48,162 message types defined in 12,183 protocol buffers models across the Google code tree.

Example 4. Ruby is a dynamically typed, interpreted, and object-oriented programming language. Due to its flexibility, it is often used to implement DSLs. Ruby on Rails is the most well-known framework implemented in Ruby. It is a web-application framework that gathers information from the web server and the database and uses it to render web pages and interact with users. Like in most web frameworks, the key element of a Ruby on Rails application is a data model expressed in an internal DSL. See Fig. 1.5 for an example. These models are used to scaffold the application using powerful code generators, as well as to access the database while it is operated. In Ruby on Rails we find examples of relational modeling, UI modeling, use of specialized editors for domain-specific models, and modeling of user interfaces.

With DSLs, we express software design using concepts from problem domain, abstracting from the implementation technology [56]. Instead of classes and loops, we talk about business entities, cash-flow processes, and customers. This allows for software to be customized by domain experts who are not programmers. For instance, many enterprise systems, implemented by skilled engineers who know about software architectures and programming, can still be customized by business-domain consultants with expertise in enterprise architecture and business processes. Similarly, many computer games support end-user extensions through various "mod" packages implemented as domain-specific programs. But MDSE can help skilled programmers, too. Protocol Buffers and Rails are *not* targeting domain experts. They increase reliability and productivity by raising the abstraction level and enabling code reuse. In both these languages models are mixed with code. Modeling is not in opposition to programming, but simply a more efficient way to program selected aspects of systems.

A common misunderstanding is that abstract models cannot be used effectively in software production, as they contain too little information for systems to be generated. In all the above examples, automatic infrastructures complete the abstract models with concrete information, effectively turning them into programs. MDSE combines two sources of information: a *model* and its *language* (captured in a language implementation). A good *language* captures the commonality in the domain, and allows the aspects that vary to be specified in *models*. KNIME, Scratch, and Protocol Buffers are extremely simple languages, yet one can derive complex systems from their models.

1.2 Model-Driven Software Engineering in Industry

Recent decades have seen an increasing interest in modeling and MDSE among practitioners and researchers. A large number of case studies and individual industrial experiences are available. These publications typically explain the domains and the circumstances in which MDSE has been successfully applied, the usage and role of models, as well as the perceived benefits, risks, and challenges encountered. Works and presentations by established researchers summarize experiences of collaboration with industry. For instance, Selić [55] mentions over seventy papers that report on industrial experiences with MDSE. Already twenty years earlier, Deursen and Klint [15] discussed the role of MDSE in industrial practice, its benefits, along with the associated risks and mitigations.

Two substantial empirical works study how industrial practitioners use MDSE: Forward and Lethbridge [19] ask 113 practitioners for attitudes towards modeling and plain code-oriented development; Hutchinson et al. [31] investigate the benefits of MDSE and attitudes towards it by surveying

Terms, Acronyms, and Buzzwords of Model-Driven Software Engineering

Beginners often struggle with the many acronyms and synonyms used for similar concepts in this field. We use the terms MDSE (Model-Driven Software Engineering) and DSL (Domain-Specific Language) to denote the software development method and the main instrument respectively.

MDSE is an umbrella term for the whole field of using models to *engineer* software. *Engineering* comprises not only the development of code—known then as MDSD (Model-Driven *Software* Development)—but also other activities, such as evolution and quality assurance. MDE (Model-Driven Engineering) and MDD (Model-Driven Development) correspond to MDSE and MDSD, but are not restricted to software, and may concern the engineering or development of other assets, such as hardware. MDA (Model-Driven Architecture) is often used synonymously with all these MDSE-related terms. However, originally MDA referred to a specific standard established by the Object Management Group [46] describing a software-design process starting with domain modeling in order to obtain platform-specific models that can ultimately be executed [22, 43].

Models in MDSE are defined in a language, which is often a DSL (Domain-Specific Language). A DSL, as opposed to a GPL (General-Purpose Language, such as Java, C#, or Scala), focuses on expressing concepts in a specific domain. DSLs should be understandable by a domain expert; their strength is their reduced expressiveness compared to GPLs. DSML (Domain-Specific Modeling Language) refers to DSLs used specifically for modeling. As for DSLs it is hard to draw the line between models and programs, the terms DSL and DSML are often used interchangeably, reflecting more the convictions of the speaker than differences of meaning. Further subsets of DSLs are domain-specific markup languages (e.g., XHTML, MathML) and domain-specific programming languages (e.g., Perl, shell-script languages).

New terms have gained popularity recently, notably Low-Code Platforms and Non-Code Platforms. They refer to a software-engineering method and a business model for rapid application development that conceptually relies on MDSE. The focus is on leveraging external DSLs with graphical syntaxes in software tools (the low-code or non-code platform) usable by end-users for generating the desired applications [54, 50, 28]. Prominent examples are Google's AppSheet, Microsoft's PowerApps (both commercial), and Eclipse's OSBP (open source).

250 and interviewing 22 professionals. As many as 83 % of the 250 respondents find MDSE beneficial, while only 5 % disagree. The empirical studies are complemented by literature reviews, such as the one of Mohagheghi and Dehlen [44] who identify and consolidate 25 papers published between 2002 and 2007 reporting on industrial experiences.

Where is MDSE used?

MDSE is used in domains where complex business logic co-exists with technical details that can be abstracted, where software should be reused, or where software interacts with hardware whose characteristics need to be modeled. Having surveyed 128 practitioners, Bone and Cloutier [6] conclude that large and long-lived software projects are more likely to adopt MDSE than short-lived ones, as the additional effort would not always pay off. Torchiano et al. [60] find that modeling correlates positively with the company size: larger organizations model more. Nevertheless, there are reports that small companies benefit from adopting MDSE, too [13].

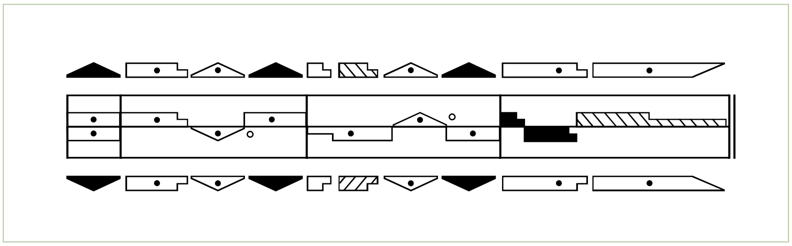


Figure 1.6: A dance choreography expressed in the visual DSL Labanotation designed by Rudolf von Laban in 1928. Source: Wikimedia Commons contributor Inigolv (CC BY-SA 4.0)

The literature review of Mohagheghi and Dehlen [44] provides documented adoption of MDSE in *telecommunication* [64, 59, 2], *business applications and financial organizations* [14], *web applications* [9], *aerospace and defense* [33], as well as *embedded* [61] *and safety-critical systems* [53]. The survey of Bone and Cloutier [6] adds *automotive* [10], *IT*, *medical* [41], and *space systems* [17]. Other substantial experience reports worth mentioning concern the domains *railway technology* [40] and *eGovernment* [42]. The latter, together with Baker, Loh, and Weil [2], are a rare source of rich and longitudinal data from companies using MDSE. Selić [55] reports similar domains, adding *industrial automation* [59] and *office automation systems* [62]. He also lists large companies that have adopted MDSE: Airbus, BAE Systems, Boeing, Lockheed-Martin, NASA-JPL, Northrop-Grumman, Raytheon, SAAB, Thales, Audi, AVL, BMW, Bosch, Carmeq, Continental, Daimler, Delphi, General Motors, Magneti Marelli, Valeo, Volvo Cars,[4] Volkswagen, ABB, Deere & Co., FMC, Siemens, Alcatel-Lucent, Ericsson, Motorola, Nortel, Siemens, UBS, and SAP. In addition to all these industrial experience reports, many publications report on commercial and non-commercial *DSLs* that have been developed for various domains, especially those listed above. The existence of these DSLs indicates that there is a demand, and likely an actual usage in practice.

Robotics is often seen as a field that can benefit well from MDSE technologies, especially from models that abstract over the hardware and low-level motion control algorithms. Robotics software is often developed in an ad hoc unsystematic way [57, 25, 11, 24, 26], and the respective control software is rarely reusable [27]. Nordmann et al. [45] identified 41 publications presenting robotics DSLs. They use a reference example of a kinematics DSL [23] developed to control robotic soccer players within the RoboCup competition.[5] Interestingly, more robotics soccer DSLs exist. CABSL [51] can be used to program the behavior of specific soccer players, such as the goalkeeper. Another example are DSLs for controlling humanoid robots, such as DANCE [30]. This textual language is inspired by Labanotation, a visual DSL invented by a German dance artist and choreographer in 1928! An example choreography is shown in Fig. 1.6.

With the advent of big-data-processing and machine-learning frameworks, whose APIs can be difficult to understand and cumbersome to use,

[4]Volvo Cars recruiters are known to ask about applicants' performance in MDSE courses.
[5]http://www.robocup.org, retrieved 2022/08

various DSLs have been presented to ease utilization of this technology. The survey by Portugal, Alencar, and Cowan [49] identifies seven DSLs, three of which were developed by Google, Microsoft, and Yahoo! to cope with their complex machine-learning and data-processing frameworks.

Finally, a classic survey by Deursen, Klint, and Visser [16] gives an overview of practice, technology, and motivation for using DSLs. Already in the year 2000, they listed over thirty DSLs documented in the literature, many in widespread use. They summarize implementation strategies, techniques, and architectures, as well as (by now mostly of historical interest) available language workbenches, which were starting to emerge at the time. A newer annotated bibliography is maintained by Lämmel [37] online.[6]

How are models used?

Almost every programmer uses models in some way for automating engineering activities, depending on what exactly is seen as a model [4]. Mohagheghi and Dehlen [44] list code generation, simulation, testing, and automatic test generation as the most frequently reported use cases. Liebel et al. [39] quantify the popularity of these use cases based on data from 112 professional developers in the embedded-systems domain. We discuss them in decreasing order of popularity according to them.

Interpretation and code generation. Software or significant parts of it can be automatically generated from models, or models can be directly executed (interpreted) as part of the system execution. In this book, we will experience interpretation and code generation not only by building examples of systems, but also by using code generation when designing and implementing DSL infrastructure. We will generate implementations of models, (de)serializers of models, model generators, tests, and editors for models, and well-formedness checkers for models. A lot of basic infrastructure of DSLs can be automatically generated using language workbenches.

Simulation. Early simulation of models is one of the most powerful techniques used by hardware and embedded-systems engineers. The construction of an executable model allows the behavior of the system to be simulated before it is actually constructed, and design mistakes to be found early. These benefits are less pronounced for software-only systems. Still, simulation makes sense for establishing properties of systems, when obtaining them directly from a running instance would be expensive or slow. For instance, virtual simulation of network protocols is much more efficient than setting up a physical network infrastructure, deploying the implementations, and running tests. Simulation makes sense for complex performance properties of many systems, as performance simulations can often be run much faster than the system can be observed in real time.

Models provide useful oracles and visualizations for system monitoring and debugging. Simulation can be used to mock components that are not yet

[6] https://github.com/slebok/yabib, retrieved 2022/08

implemented, or to test implemented components to see whether they behave as expected. Errors are flagged whenever the actual execution diverges from the specification given in a model. When program state, or problematic data, are visualized as models, debugging any potential divergence from the specification is easier. From language-engineering perspective, simulation and runtime-monitoring are both special use cases of interpretation.

A related activity to simulation is *instance generation*. Instances of data models can be generated to serve as test data. Many design mistakes in data models can be established quickly by analyzing examples of unexpected instances of data, cases which should be disallowed [32].

Documentation and information. Models provide excellent documentation, given their proximity to the domain. Often, they are self-documenting [16] and can be directly used as documentation, or embedded into such, regardless of whether models have a textual or graphical syntax. Likewise, models foster conversations and coordination among different roles, including non-technical ones, such as domain or sales experts. While developers can interact talking about code, using models can ease these interactions. However, documentation and information, even though reported as the third most frequent usage by Liebel et al. [39], is never the prime usage in MDSE, which aims at using models in automation.

Model checking and verification. System models can be verified, for example, for safety properties. Combined with code generation or interpretation, model checking and verification of models significantly raise our confidence that properties of the model are also properties of the final system. Liebel et al. [39] study usage of model checking and verification at a fine-granularity level. They distinguish, most frequent first: structural consistency checks, behavioral consistency checks, timing analysis, formal verification, safety compliance checks, and reliability analysis. Tools for model checking and verification are available predominantly for established languages (such as MATLAB Simulink), since developing verification infrastructure is relatively expensive and requires advanced expertise. Such tools rarely exist for project-specific languages. Consistency checks are considerably easier and we will be discussing them regularly throughout the book.

Test-case generation. If models (like finite state machines, one of the running examples in this book) describe the possible behaviors of a system, we can use them to generate test cases. Given possible sequences of states from a model, one can generate test input data that ensures high test coverage for the code-under-test and can be used to check whether the behavior of the implementation adheres to the model. An explicit model, created independently of the code, is a trustworthy representation of the designer's intention. Therefore it allows testing against these intentions. This is in contrast to test cases derived from the possibly buggy code-under-test.

Traceability. Large organizations, especially in safety-critical domains, need to establish traceability links between engineering artifacts. Most often, such links are necessary between requirements and code. Traceability information allows the completeness of the implementation to be checked against the requirements, analysis of the impact of (maintenance or evolution) changes to the system, and tracing of bug reports. In safety-critical domains, such as aerospace and automotive, traceability is often prescribed by a safety standard. Specific trace models, but also many other models, can be used to record and exploit traceability information.

Model-based system integration. Finally, let us mention a use case that was not directly listed by Liebel et al. [39], but which is common as well: using models for integrating systems. Specifically, if systems rely primarily on models, then data exchange and integration of systems can be done via models. This is particularly convenient, since models can be translated to other languages by *model transformations*, which are small programs implemented in languages specialized for model transformation.

What are the benefits of MDSE?

Now that we know how models are used, let us discuss what an organization can gain by adopting MDSE. We again discuss the benefits ordered by the frequency reported by Liebel et al. [39].

Improved quality. The benefit of MDSE reported most frequently by professionals is the improvement of quality [39]. First, generated code is typically of high quality. A substantial effort is put into the design of a generator, and any bug fixed there immediately improves the quality of all the generated code, reducing errors for all users. Second, simulation allows errors to be detected and fixed early, and to exercise more system behavior than could be done in physical tests. Third, models improve the quality of requirements in the sense that some requirements can be expressed within the model [5], which allows errors in requirements to be found, consistency and completeness of requirements to be checked, and the clarity of requirements to be enhanced.

Improvements of quality are also confirmed by Kieburtz [34] who observes a reduction of error rates and productivity gains in a controlled experiment. Mohagheghi and Dehlen [44] find in their survey that, in addition to errors being found early, fewer code inspections were necessary in published case studies of MDSE. In the words of Baker, Loh, and Weil [2], in the context of a case study at Motorola: "it is not unusual to see a 30X—70X reduction in the time needed to correctly fix a defect detected during system integration testing. This reduction is attributed to the ability to add a model test that illustrates the problem, fix the problem at the model level, test the fix by running a full regression test suite on the model itself, regenerate the code from scratch, and run the same regression test suite on the generated code."

Improved reusability. Reuse is the ability to take an existing piece of software and modify it to fit another purpose or a new context [16]. For instance, you may want to reuse code written for specific hardware for different hardware. Another published case study from Motorola reports "reuse of designs and tests between platforms or releases" as a significant benefit [64]. Instead of copying and modifying code, the idea is that the modifications are represented in the models, so you account for changes, incorporate them in the language, and then when you want to have a different system, you modify the model, re-generate code or just run the interpreter. It is just easier to specify models than to modify code [56]. Models foster knowledge conservation and reuse [16]. Furthermore, since models typically abstract over hardware, one can write different generators or interpreters for different hardware platforms: model-driven software is more easily *retargetable*. Many modeling languages, known as *variability-modeling languages*, have been specifically designed to support reuse. We return to them in Chapters 12 and 13.

Improved reliability. Improved reliability is a consequence of automation and of the reuse of expert knowledge in code generation or interpretation [16]. Selić [56] says "[...] modern optimizing compilers can outperform most practitioners when it comes to code efficiency. Furthermore, they do it much more reliably." That generated code is more reliable was also clearly shown in the controlled experiment of Kieburtz et al. [35]. Reliability was also a prime benefit observed in the Motorola case study of Weigert and Weil [64] as (i) insecure or unreliable coding practices can be avoided, (ii) specific secure coding policies and patterns can be enforced, (iii) problems related to reliability can be detected early (in the code generator implementation or in models), and (iv) separation of concerns can help in assessment of reliability.

Improved traceability. Models themselves can be used for establishing and exploiting traceability, but already by using MDSE, traceability is obtained as a by-product [66]. Especially when transforming models into other models, model-transformation engines produce traces automatically; they create and maintain a trace model, which can be queried. When requirements are expressed as part of the model, traceability is naturally improved. Finally, models foster the comprehension of change impacts when the system is changed [15], which is also a traceability-related improvement. theo

Improved maintainability. Multiple authors report maintainability and productivity gains as prime benefits of MDSE [16, 56, 15, 35, 19, 31]. Software defined in domain terms is easier to maintain. Models are easier to understand than low-level code, and can serve the role of documentation at times. It is easier to train new developers to tailor systems using an abstract DSL instead of low-level code, since most DSLs ensure that the changes stay within assumed design invariants. Collaboration and coordination among developers is improved through models [19]. MDSE allows easier modifications [19] and comprehension of change impacts. Models and their languages more explicitly represent domain-specific knowledge, which is also represented in a platform-independent manner [15]. The latter also enhances *system portability* [31, 29].

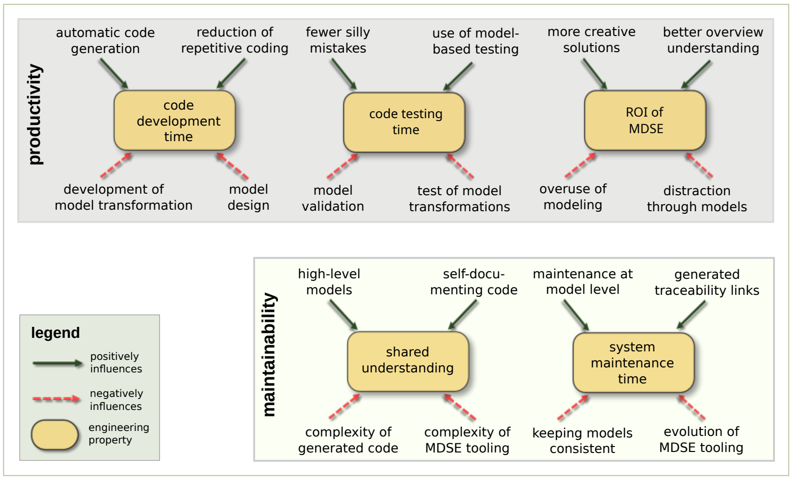

productivity
automatic code generation
reduction of repetitive coding
fewer silly mistakes
use of model-based testing
more creative solutions
better overview understanding
code development time
code testing time
ROI of MDSE
development of model transformation
model design
model validation
test of model transformations
overuse of modeling
distraction through models
legend
positively influences
negatively influences
engineering property
maintainability
high-level models
self-documenting code
maintenance at model level
generated traceability links
shared understanding
system maintenance time
complexity of generated code
complexity of MDSE tooling
keeping models consistent
evolution of MDSE tooling

Figure 1.7: *Illustrative influences of MDSE. Adapted from Hutchinson et al. [31] and omitted 'system portability'. ROI = return on investment*

Improved productivity. The reasons for better productivity are mostly the same as for improved maintainability: better comprehension thanks to abstraction and domain-orientation. Various roles, including domain experts, can understand the models [16, 56]. Systems can be created faster; sometimes they can even be instantiated by non-technical domain experts, who create the models and then initiate code generation with automatic deployment [56]. Such systems are more likely to be usable and to meet the original requirements [19]. Huge productivity gains are quoted by practitioners of MDSE in interviews [31]: at least two-fold, but even eight-fold! Interestingly, some interviewees mention that the increases might be hidden from management to protect against budget cuts.

What are the risks of MDSE?

MDSE is, of course, not a panacea for all kinds of organizations, projects, and domains. All the benefits discussed above are affected by various negative forces. Figure 1.7 summarizes examples of factors that influence various aspects of productivity (e.g., code development time) and maintainability according to Hutchinson et al. [31]. Both positive and negative influences should be taken into account when assessing MDSE. In practice, depending on the project context, certain aspects will outweigh others.

Return on investment. MDSE requires additional effort for engineering systems (including maintenance). There is a risk that this effort is too high and will not pay off when models are not useful enough [60, 16]. Another challenge can be the costs for education and training that are necessary for adopting MDSE [39, 16].

Half-baked adoption. If the potential of MDSE is not fully exploited, especially if models end up being solely used for documentation, the benefits are easily lost. Documentation-only models quickly diverge from reality, when they are not used for automation. Developers lose interest and do not maintain them, and the project gives up using models altogether [56].

MDSE and Agile Software Engineering

MDSE is often (wrongly) associated with plan-based software engineering, requiring heavy upfront investment and planning. In contrast, agile software engineering promotes incrementality, continuous integration, delivery, and continuous deployment [18]. In our experience, these practices are perfectly applicable to development of DSLs and systems using DSLs. Furthermore, key agile practices, such as automated testing and continuous integration, require extreme automation, and in that synergize with MDSE practices. MDSE can accelerate programming in agile projects, and can enable automation for continuous test, integration, and deployment. The MDSE trend is further seen in the rise of low-code and no-code platforms (see the box on p. 12).

Model and language quality. Selić [56] sees low-quality models and non-adequate abstraction levels as significant risks. Since MDSE is so powerful, low-quality models can have far-reaching negative impact. Deursen, Klint, and Visser [16] see balancing between generality and domain-specificity in DSL development as a challenge. An important quality property for DSLs is that they are properly scoped—so neither include too many nor too few concepts of the domain. A badly scoped DSL is a risk. Finally, when code generation is used, the efficiency of the generated code is a considered risk [16], but as we point out above, generated code is often more efficient.

Model consistency. Models need to be kept in sync with code and other artifacts. A risk of inconsistencies arises when consistency is not actively maintained [19]. Forward and Lethbridge postulate a need for better traceability and facilities for partial updates or co-evolution. Use of embedded DSLs (within code) could also lower this risk.

Tooling. Quality of tools is definitely a problem, even though the situation has improved significantly over the last decade. Hutchinson et al. [31] report over 50 tools used by the respondents, which suggests a lack of maturity—definitive market leaders are yet to emerge. Tools are immature; complaints about prices are common. Liebel et al. [39] further emphasize tool interoperability and tool usability. In this book however, we use solely open-source tools and widely available programming languages.

Bug fixing. While with MDSE introducing changes is easier, bug fixing may be perceived as harder [19]. This is confirmed by our own experience, especially in teaching, when students are new to the subject. Experience helps; many problems are adoption related. The idiosyncrasies of many tools challenge users early on. One has to remember though that when software is developed without MDSE, the number of bugs is higher, even if some of them appear very simple.

Further Reading

The idea of DSLs is usually tracked to the seminal paper of Landin [38] on the next “700 programming languages.” Landin is concerned with a family of related languages, where differences are introduced by (possibly significant amounts of) syntactic sugar, rather than with creating special purpose languages. His languages differ syntactically, but share the same expressiveness. The suggestive title, and the fact that it argues the need for diverse language syntaxes for various needs, is probably the reason why this paper is considered as the first mention of DSLs.

The book by Stahl and Völter [58] is likely the most-referenced book on MDSE that really helped to establish MDSE as a field and made it known to practitioners. It presents the UML-based approach to DSLs. Using UML and stereotyping was one of the earliest and remains one of the easier ways to create graphical languages. In his newer book, Völter [63] focuses on using and developing DSLs. In many ways, this text is more comprehensive than ours, however we strive to present the material in style and structure suitable for use in an academic course, without assuming an extensive training in compiler theory. A classic presentation of DSL design is given by Fowler and Parsons [21], who thoroughly and excellently discuss the patterns and guidelines for implementing and using DSLs. However, their implementation of DSLs is not as much model-driven as in this book, which focuses on automation more heavily. Another recent text on MDSE is the book of Combemale et al. [12]. It teaches language design for MDSE in a concrete manner, showing models and code, and discussing examples. It includes exercises and code in a git repository, encouraging experimentation. The book gives a good coverage of language workbenches and of external DSL design. It also brings in some formal, mathematical semantics to the reader to mitigate a bit the vagueness found in some other MDSE literature, however it remains strictly in the object-oriented universe, while we are trying to bridge the gap between the functional and object-oriented DSL traditions.

The book of Lämmel [36] is the most recent addition to this body of knowledge. The book covers a breadth of methods with technologies and theories for the implementation of languages, from both an object-oriented and a functional perspective, whereas the latter prevails. Haskell, Python, and Java are used as the key programming languages. The object-oriented approach is presented directly in Java, with limited use of modeling and no explicit domain analysis. The book is strong on the programming language and semantics background (e.g., it introduces Lambda calculus and abstract interpretation), and comes with a code repository and many exercises.

Brambilla, Cabot, and Wimmer [8] present a good overview of model-driven development architectures, processes, and benefits, in a manner very suitable for experienced software developers who appreciate the software engineering issues solved by MDSE, and who are trained in language design and implementation.

Rogers and Girolami [52] give an efficient introduction to data analytics and machine learning—this helps one to appreciate the KNIME example in Fig. 1.2.

References

[1] Charles Babbage. *Note on the application of machinery to the computation of astronomical and mathematical tables*. 1822 (cit. p. 6).

[2] Paul Baker, Shiou Loh, and Frank Weil. “Model-driven engineering in a large industrial context–Motorola case study”. In: *International Conference on Model Driven Engineering Languages and Systems (MODELS’05)*. 2005 (cit. pp. 13, 16).

[3] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “A study of variability models and languages in the systems software domain”. In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640 (cit. p. 7).

[4] Jean Bézivin. “On the unification power of models”. In: *Software and System Modeling* 4.2 (2005), pp. 171–188 (cit. pp. 8, 14).

[5] Stefan Biffl, Richard Mordinyi, and Alexander Schatten. “A model-driven architecture approach using explicit stakeholder quality requirement models for building dependable information systems”. In: *Proceedings of the 5th International Workshop on Software Quality*. IEEE Computer Society. 2007, p. 6 (cit. p. 16).

[6] Mary Bone and Robert Cloutier. “The current state of model based systems engineering: results from the OMG SysML request for information 2009”. In: *Proceedings of the 8th Conference on Systems Engineering Research*. 2010 (cit. pp. 12, 13).

[7] George Box and Norman Draper. *Empirical Model-Building and Response Surfaces*. Wiley, 1987 (cit. p. 5).

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012 (cit. p. 20).

[9] Marco Brambilla, Stefano Ceri, Piero Fraternali, Roberto Acerbis, and Aldo Bongio. “Model-driven design of service-enabled web applications”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. ACM, 2005 (cit. p. 13).

[10] Manfred Broy. “Challenges in automotive software engineering”. In: *Proceedings of the 28th International Conference on Soft. Eng. (ICSE’06)*. ACM, 2006 (cit. p. 13).

[11] Davide Brugali. *Software engineering for experimental robotics*. Vol. 30. Springer, 2007 (cit. p. 13).

[12] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge Into Tools*. CRC Press, 2016 (cit. p. 20).

[13] Jesús Sánchez Cuadrado, Javier Luis Cánovas Izquierdo, and Jesús García Molina. “Applying model-driven engineering in small software enterprises”. In: *Sci. Comput. Program.* 89.PB (Sept. 2014), pp. 176–198 (cit. p. 12).

[14] Gan Deng, Tao Lu, Emre Turkay, Aniruddha Gokhale, Douglas C Schmidt, and Andrey Nechypurenko. “Model driven development of inventory tracking system”. In: *Proceedings of the OOPSLA 2003 Workshop on Domain-Specific Modeling Languages*. 2003 (cit. p. 13).

[15] Arie Deursen and Paul Klint. *Little languages: little maintenance?* Tech. rep. Amsterdam, The Netherlands, 1997 (cit. pp. 11, 17).

[16] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: An annotated bibliography”. In: *SIGPLAN Notices* 35.6 (2000) (cit. pp. 14–19).

[17] Harald Eisenmann, Juan Miro, and Hans Peter Koning. “MBSE for European space-systems development”. In: *INSIGHT* 12.4 (2009), pp. 47–53 (cit. p. 13).

[18] Brian Fitzgerald and Klaas-Jan Stol. “Continuous software engineering and beyond: trends and challenges”. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. 2014 (cit. p. 19).

[19] Andrew Forward and Timothy C. Lethbridge. “Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals”. In: *Proceedings of the 2008 International Work-*

shop on Models in Software Engineering. MiSE ’08. 2008 (cit. pp. 11, 17–19).

[20] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004 (cit. p. 8).

[21] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley, 2011 (cit. p. 20).

[22] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003 (cit. p. 12).

[23] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. “A domain specific language for kinematic models and fast implementations of robot dynamics algorithms”. In: *Workshop on Domain-Specific Languages and Models for Robotic Systems*. 2011 (cit. p. 13).

[24] Sergio Garcia, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Rebekka Wohlrab. “An architecture for decentralized, collaborative, and autonomous robots”. In: *International Conference on Software Architecture (ICSA)*. 2018 (cit. p. 13).

[25] Sergio Garcia, Daniel Strueber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. “Robotics software engineering: A perspective from the service robotics domain”. In: *28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2020 (cit. p. 13).

[26] Sergio Garcia, Daniel Strueber, Davide Brugali, Alessandro Di Fava, Patrizio Pelliccione, and Thorsten Berger. “Software variability in service robotics”. In: *Empirical Software Engineering* (2022) (cit. p. 13).

[27] Sergio Garcia, Daniel Strueber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. “Variability modeling of service robots: Experiences and challenges”. In: *13th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 2019 (cit. p. 13).

[28] Dre Hendriks. “The selection process of model based platforms”. MA thesis. Radboud University Nijmegen, 2017 (cit. p. 12).

[29] R. M. Herndon Jr. and V. A. Berzins. “The realizable benefits of a language prototyping language”. In: *IEEE Trans. Softw. Eng.* 14.6 (June 1988), pp. 803–809 (cit. p. 17).

[30] Liwen Huang and Paul Hudak. *Dance: A declarative language for the control of humanoid robots*. Tech. rep. Department of Computer Science, Yale University New Haven, CT, USA, 2003 (cit. p. 13).

[31] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. “Empirical assessment of MDE in industry”. In: *ICSE*. 2011 (cit. pp. 8, 11, 17–19).

[32] Daniel Jackson. *Software Abstractions*. MIT Press, 2006 (cit. p. 15).

[33] Eric Jouenne and Véronique Normand. “Tailoring IEEE 1471 for MDE support”. In: *UML Modeling Languages and Applications*. Ed. by Nuno Jardim Nunes, Bran Selić, Alberto Rodrigues da Silva, and Ambrosio Toval Alvarez. Springer-Verlag, 2005 (cit. p. 13).

[34] Richard B. Kieburtz. *Defining and Implementing Closed, Domain-Specific Languages*. Invited talk at the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG). 2000 (cit. p. 16).

[35] Richard B. Kieburtz et al. “A software engineering experiment in software component generation”. In: *ICSE*. IEEE Computer Society, 1996 (cit. p. 17).

[36] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018 (cit. p. 20).

[37] Ralf Lämmel. *Yet another annotated SLEBOK bibliography*. 2014. URL: https://github.com/slebok/yabib (cit. p. 14).

[38] Peter J. Landin. "The next 700 programming languages". In: *Commun. ACM* 9.3 (1966), pp. 157–166 (cit. p. 19).

[39] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. "Assessing the state-of-practice of model-based engineering in the embedded systems domain". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2014 (cit. pp. 8, 14–16, 18, 19).

[40] Anthony MacDonald, Danny Russell, and Brenton Atchison. "Model-driven development within a legacy system: An industry experience report". In: *Australian Software Engineering Conference*. 2005 (cit. p. 13).

[41] Amen Ra Mashariki, LeeRoy Bronner, and Peter Kazanzides. "Designing and developing medical device software systems using the model driven architecture (MDA)". In: *Proceedings of the 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability*. HCMDSS-MDPNP '07. 2007 (cit. p. 13).

[42] Niklas Mellegård, Adry Ferwerda, Kenneth Lind, Rogardt Heldal, and Michel RV Chaudron. "Impact of introducing domain-specific modelling in software maintenance: an industrial case study". In: *IEEE Trans. on Soft. Eng.* 42.3 (2016), pp. 245–260 (cit. p. 13).

[43] Stephen J. Mellor. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 2004 (cit. p. 12).

[44] Parastoo Mohagheghi and Vegard Dehlen. "Where is the proof?—a review of experiences from applying MDE in industry". In: *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*. ECMDA-FA'08. 2008 (cit. pp. 12–14, 16).

[45] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. "A survey on domain-specific modeling and languages in robotics". In: *Journal of Software Engineering in Robotics (JOSER)* 7.1 (2016), pp. 75–99 (cit. p. 13).

[46] Object Management Group. *MDA Guide revision 2.0*. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01. 2014 (cit. p. 12).

[47] Object Management Group. *Unified Modeling Language Specification 2.5.1*. https://www.omg.org/spec/UML. 2017 (cit. p. 8).

[48] Marian Petre. "UML in practice". In: *Proceedings of the 2013 International Conference on Soft. Eng.* IEEE Press. 2013 (cit. p. 8).

[49] Ivens Portugal, Paulo S. C. Alencar, and Donald D. Cowan. "A survey on domain-specific languages for machine learning in big data". In: *CoRR* abs/1602.07637 (2016). arXiv: 1602.07637 (cit. p. 14).

[50] Clay Richardson and John R Rymer. "Vendor landscape: the fractured, fertile terrain of low-code application platforms". In: *FORRESTER, Janeiro* (2016) (cit. p. 12).

[51] Thomas Röfer. "CABSL—C-based agent behavior specification language". In: *RoboCup 2017: Robot World Cup XXI*. Lecture Notes in Artificial Intelligence. Springer, 2018 (cit. p. 13).

[52] Simon Rogers and Mark Girolami. *A First Course in Machine Learning*. 2nd Edition. Chapman & Hall/CRC, 2016 (cit. p. 20).

[53] Laurent Safa. “The practice of deploying DSM, report from a Japanese appliance maker trenches”. In: *6th OOPSLA Workshop on Domain Specific Modeling*. 2006 (cit. p. 13).

[54] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. “Supporting the understanding and comparison of low-code development platforms”. In: *2020 46th Euromicro Conference on Soft. Eng. and Advanced Applications (SEAA)*. IEEE. 2020 (cit. p. 12).

[55] Bran Selić. *Model-Based Software Engineering in Industry: Revolution, Evolution, or Smoke?* https://www.youtube.com/watch?v=miPZyfRIcs8. 2017 (cit. pp. 11, 13).

[56] Bran Selić. “The pragmatics of model-driven development”. In: *IEEE Software* 20.5 (2003), pp. 19–25 (cit. pp. 5, 7, 8, 11, 17–19).

[57] SPARC. *Robotics 2020 Multi-Annual Roadmap*. 2016. URL: https://old.eu-robotics.net/cms/upload/topic_groups/H2020_Robotics_Multi-Annual_Roadmap_ICT-2017B.pdf (cit. p. 13).

[58] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005 (cit. p. 20).

[59] Miroslaw Staron. “Adopting model driven software development in industry: a case study at two companies”. In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. MoDELS’06. 2006 (cit. p. 13).

[60] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. “Preliminary findings from a survey on the md state of the practice”. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2011 (cit. pp. 12, 18).

[61] Bruce Trask, Dominick Paniscotti, Angel Roman, and Vikram Bhanot. “Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 2006 (cit. p. 13).

[62] Nikola Trčka, Martijn Hendriks, Twan Basten, Marc Geilen, and Lou Somers. “Integrated model-driven design-space exploration for embedded systems”. In: *Embedded Computer Systems (SAMOS), 2011 International Conference on*. IEEE. 2011, pp. 339–346 (cit. p. 13).

[63] Markus Völter. *DSL Engineering. Designing, Implementing and Using Domain Specific Languages*. 2013. URL: http://www.dslbook.org (cit. p. 20).

[64] Thomas Weigert and Frank Weil. “Practical experiences in using model-driven engineering to develop trustworthy computing systems”. In: *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*. Vol. 1. IEEE. 2006, 8–pp (cit. pp. 13, 17).

[65] Jon Whittle, John Hutchinson, and Mark Rouncefield. “The state of practice in model-driven engineering”. In: *IEEE software* 31.3 (2014), pp. 79–85 (cit. p. 8).

[66] Stefan Winkler and Jens von Pilgrim. “A survey of traceability in requirements engineering and model-driven development”. In: *Software & Systems Modeling* 9.4 (2010), pp. 529–565 (cit. p. 17).

Everything is a model.
Jean Bézivin [8]

2 Building Modeling Languages

Our goal is to automate the development of software in a given domain by using models to describe its essential characteristics and producing applications using code generation and interpretation. To this end, we need a *language* to express the models, its *instances*. This book is about designing and implementing languages—your opportunity to become a language designer. In this chapter, we first discuss the need for domain specificity, and then follow with a brief overview of the topics involved in language design—all of them discussed in great detail in later chapters.

2.1 The Story of Abstraction in Programming Languages

General-purpose programming languages (GPLs) aim at describing algorithms and structuring software systems. As such, they are not tailored towards a particular domain, but contain general computation-related concepts such as *loops*, *conditionals*, and *types*. Over time, GPLs have become increasingly abstract, to tackle ever-increasing complexity of software.

The very first programming languages—machine languages—only contained instructions that the machine's processor could execute directly, such as *counting*, *reading registers*, and basic *input/output* operations on memory and hardware devices. The instructions were stored as binary numbers. Control flow jumps in these languages used concrete memory addresses to indicate program locations. In order to allow more complex programs to be written, the assembly languages were designed; colloquially known as assemblers [9]. They brought *jumps* and limited *arithmetic expressions*. Numeric instruction codes were replaced by human-readable mnemonic names and named labels for jumps were added. The introduction of assembly language dramatically raised the level of abstraction in programs, hiding the complex low-level aspects of the machine. For the first time, a compiler had to be used to transform the assembly code into machine-executable binary code.

The next generation of languages were touted as being high-level. Fortran was the first [4], Algol 60 followed shortly after [3]. Both were imperative languages, introducing concepts such as *loops*, *conditionals*, and *recursion*. LISP, presented around the same time, was the first functional language based on Church's lambda calculus with the idea of *function* value as the main building block of a program [39]. Finally, COBOL brought abstractions needed for business programming, such as *macros* and *hierarchical data structures* [15]. All high-level languages shared the goal of moving the computation away from low-level aspects of machine operation to abstract terms representing application-level concepts.

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_2

The next well-recognized evolutionary leap in programming languages was the introduction of object-oriented programming. Simula 67 extended Algol 60 with strong modularization concepts: *objects* and *classes*, *specialization* and *generalization* of data types, *aggregation* of related data, and *encapsulation* of related behavior [18]. Among the most popular object-oriented languages were C++, Java, and C#—one of which you are probably familiar with. Nowadays, we observe an increasing incorporation of functional-programming concepts, such as *anonymous functions* (also known as *lambdas*) and *closures* into object-oriented languages, as can be seen with Java 8, Kotlin, Python, etc. Scala is probably the language combining the most object-oriented and functional features.

In retrospect, the history of programming languages is a history of abstractions. The later languages enrich the abstraction capabilities of their predecessors, even though the predecessors were already Turing-complete. This means that, for a long time now, the expressiveness of a programming language has been much less important than its abstraction capabilities.

2.2 The Ultimate Abstraction: Domain-Specific Languages

A practical way to further increase the abstraction in software development is to introduce domain-specific languages. With DSLs, one can add domain-specific concepts directly into the language. Instead of trying to find a general-purpose abstraction, one can design abstractions that capture the essential aspects of the domain and hide the inessential ones. Such a language can no longer be used for arbitrary applications, but it raises the abstraction level for applications in the problem domain that it supports. DSLs are the ultimate abstraction [27].

Good examples of successful languages that abstract away substantial amounts of details, yet put a lot of power into the hands of their users, are HTML and SQL. Both hide the complex algorithms needed to execute them. HTML hides the layout algorithms. SQL abstracts away query execution mechanisms. Both expose high-level domain-specific concepts, for instance, document elements and data relationships.

Definition 2.1. *A* domain-specific language (DSL) *is a computer programming or modeling language of limited expressiveness focused on a particular domain or its aspect.*[1]

Today, engineers can use modern tools, the so-called *language workbenches*, to create languages for much smaller domains than SQL and HTML. Language workbenches allow DSL infrastructure to be built cost-effectively. The next leap in the history of abstraction in programming is in the hands of software architects. It is now feasible to create compilers solely to increase the quality and efficiency of individual software projects and products.

The concepts within a DSL need to pertain to a domain to be coherent enough and understandable for the users. Otherwise, it would be too difficult

[1] After Fowler and Parsons [23]

to use the language. In fact, limiting the language elements in a DSL to a domain is one of the core strengths of DSLs, which eases their use and makes them appealing to both domain experts, end-users knowledgeable in the domain, and of course developers. A typical definition of domain is as follows.

Definition 2.2. *A* domain *is an area of knowledge scoped to maximize the satisfaction of the requirements, including a set of concepts and terminology understood by practitioners in the area, and including knowledge of how to build software systems (or their parts) in the area.*[2]

The need to build a new language introduces a tension into MDSE: to lower the cost for system engineering we need to invest additional development into language engineering. Two forces act to reduce this cost: distribution of cost across multiple projects, and advances in language technology.

Distribution of cost across multiple projects. MDSE is cost-effective if the DSL is reused across multiple projects. For instance, a consulting firm that customizes enterprise-resource-planning systems can build and maintain extensions for several different customers efficiently using a DSL. A DSL for coordinating autonomous robots will benefit the developers if they can reuse the same language in several similar robotics projects. MDSE is beneficial within a single project if concise domain-specific models replace substantial boilerplate code in many places, as is the case with embedded SQL or LINQ for database access. In all these examples, the cost of language development is offset by claiming the benefits multiple times.

Advances in software tools. Any organization deciding to develop a DSL needs language designers. Traditionally, this was not an easily available skill. Language designers and implementers were rare, and they found jobs in compiler companies. However, thanks to the emergence of well-integrated *language workbenches* along with the inclusion of meta-programming (cf. Chapter 7) facilities and libraries in mainstream GPLs, implementing DSLs has become much easier. It no longer requires specialized compiler knowledge or excessive time. It can be undertaken by most software developers who have completed a software-engineering or computer-science education, without dominating the cost of the primary development.

Definition 2.3. *A* language workbench *is a tool for creating and using (domain-specific) languages.*

Language workbenches are a rather mature technology, that has existed already since 1980 [20]. One example is Xtext, a modern tool for development of *textual* DSLs. We discuss it in Chapter 4. A popular tool for development of *graphical* DSLs is Sirius [59, 61]. The gallery of languages developed using Sirius is worth a check.[3] Figure 2.1 shows a

[2]After Czarnecki and Eisenecker [17]

[3]https://www.eclipse.org/sirius/gallery.html, retrieved 2022/08

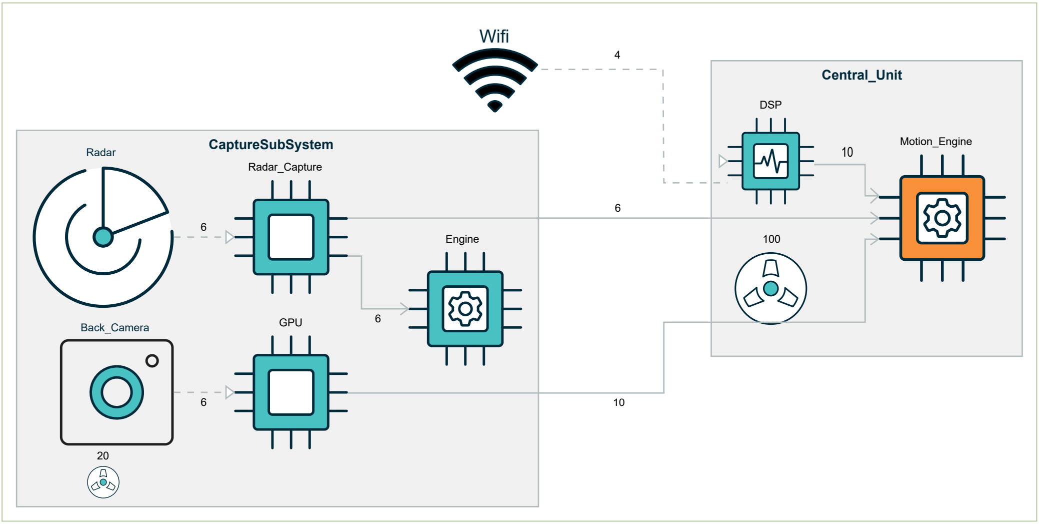


Figure 2.1: A robot-flow (architecture) model created in a graphical DSL designed in the language workbench Sirius Web, the cloud-computing web-based version of Eclipse Sirius, supporting building live collaborative editing environments for graphical DSLs

robot architecture diagram in a visual DSL designed with a web-based variant of Sirius.[4] The standard graphical editor for Ecore models, used to create many figures in this book, is implemented using Sirius, too.

Projectional language workbenches are conceptually similar to the workbenches for graphical languages, but support both textual and graphical syntax in an integrated manner. They achieve this by replacing a standard text editor with a so-called projectional editor. In contrast to a regular text editor combined with a parser, a projectional editor allows users to edit a structured tree representation of the model directly. The structure is rendered into a textual notation which creates an illusion of a text editor, very much like WYSIWYG editors for HTML or for word processing. A user editing the text directly changes the nodes of the underlying representation, without any parsing. This technology allows for rendering of graphical notations and providing both textual and graphical notations within a single language, even within a single document. We return to projectional languages and projectional editing in Chapter 13.

2.3 What Is a Language Built From?

Let us now consider what is it that we have to build when we design a new language; what is a language? Or more precisely: how can language design and implementation be split into smaller tasks? Most often we organize a language implementation in a chain of processors in a pipeline. The components are coarsely divided into two large groups, corresponding to the main parts of any language, the syntax and the semantics.

[4] https://www.eclipse.org/sirius/sirius-web.html, retrieved 2022/08

Instance of a DSL: A Model, Code, a Program, or a Mogram?

We build DSLs to write models and programs also called *instances*. Interestingly, there is no single English noun that describes the different kinds of instances, other than "instance" itself, which is rather abstract and cryptic. The instances are typically called "models," but are also referred to as "programs" or "code," even if these words do not really mean the same. In our context, it is essentially equivalent to talk about models, code, or programs. After all, almost everything in software engineering is a model [8].

Kleppe [31] tried to introduce a neologism *mogram* to describe the things that can be written in a language, emphasizing the commonalities between models, programs, and code. In her view, the most important part of a language is the definition of the abstract syntax (the meta-model). The actual name used for the instances is less important. Sadly, the word "mogram" has not caught on in the community. Völter [60] explicitly points out that he does not distinguish between model, code, and program; if he uses model and program in the same sentence, then model refers to the more abstract representation. So, abstraction is his main characteristic of a model, in line with Def. 1.1. Consequently, we also use the terms model, program, and code synonymously in this book.

Definition 2.4. *The* syntax *is the definition of the principles and processes by which sentences are constructed in a particular language.*[5]

The above definition, originally proposed for natural languages (the languages spoken by humans), applies well to programming and modeling languages. Syntax defines what programs we can write in a language. The *semantics*, on the other hand, is concerned with the meaning of the programs written in a given language.

Definition 2.5. *The* semantics *are the (study of) meanings of a language.*[6]

Similarly to syntax, the term "semantics" is applied to natural languages spoken by people. Logicians introduced it to formal languages, to talk about the meaning of terms in formal logics. The inspiration from logics led the early theoretical computer scientists to adopt the distinction between the syntax and semantics in the definition of programming languages—the distinction that carried over to modern language implementation patterns.

Example 5. Let us explore the concepts of syntax and semantics using an example. Consider a simple language for controlling mobile robots, with the uninspiring name `robot`. It loosely follows the principle of reactive control, a specific way of controlling the behavior of robots.[7] An example model can be found in Fig. 2.2. There are two key aspects that organize models in this hypothetical language: modes of operation and flows between modes—continuations. We have four modes in the example model: `RandomWalk`, `MovingForward`, `Avoid`, and `ShutDown`. The modes can be nested. The last three modes are nested in the first mode (`RandomWalk`).

A mode may contain other modes, actions, and reactions. *Actions* resemble regular programming-language statements—they are immediately executed

[5] After Chomsky [13].
[6] After Merriam-Webster's Collegiate Dictionary.

```
-> RandomWalk {
on clap -> ShutDown

-> MovingForward {
  move forward at speed 10
  on obstacle -> Avoid
}

Avoid {
  move backward for 1 s
  turn by random (-180,180)
} -> MovingForward

ShutDown { return to base }
}
```

Figure 2.2: *An example of a control model in the textual* `robot` *language, describing a robot performing a random walk while avoiding obstacles*

as the mode is activated, in the order listed. The actions in the example are: `move`, `turn`, and `return to base`. *Reactions*, introduced using the keyword `on`, are not executed immediately, but registered and suspended. Each reaction is triggered by an event, when it switches the mode to a new mode. The two events in the example are: `obstacle` and `clap`. Reactions are only active if their mode is active. Reactions are registered on the fly when a mode is activated, but are only handled after all actions are completed (non-preemptively). For instance, if the robot is in the `MovingForward` mode and encounters an obstacle, the active mode becomes `Avoid`.

A mode can also have a *continuation* mode, a successor. These are indicated using the arrow symbol (`->`). If a mode has a successor mode, then the control switches to it immediately after all actions have been executed. For instance, after `Avoid` the control moves to `MovingForward`. If a mode has no successor, the control stays in place, and awaits for any possible reaction triggers. In `robot`, one can only define a single successor for a mode. If control needs to flow to various modes as a result of execution, this can only be done by registering reactions that have different targets.

The same arrow symbol (`->`) is also placed before the initial mode, in the context of its containing mode (see `MovingForward`). There must be exactly one initial mode at each level of nesting.

The syntax of a model (or a program) is what you can directly see and read. For instance, when looking at Fig. 2.2, you see the syntax of our example model. The syntax is described with phrases of the following kind:

- *A mode may contain other modes, actions, and reactions.*
- *There must be exactly one initial mode at each level of nesting.*

The semantics of a model define what the model means: how the robot shall behave according to the model. For the model in Fig. 2.2, the semantics is that of a random walk. Semantics are defined over all instances of a language, but they are only implemented once for the language. Semantics regulate detailed aspects of behavior, for instance, whether modes are pre-emptive or not. If you specify in the semantics that modes are pre-emptive, this would mean that modes could be switched whenever a suitable reaction is triggered, leading

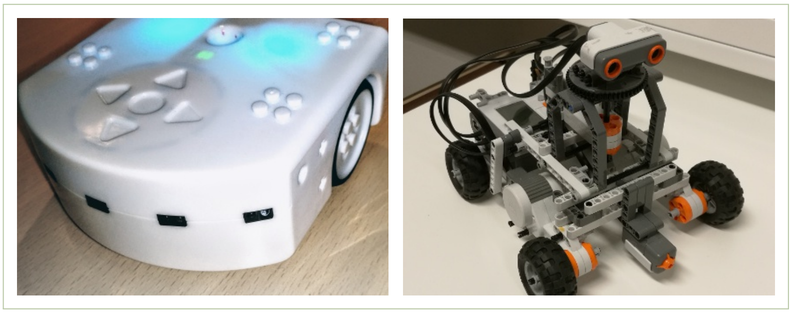


Figure 2.3: Two example educational robots that are possible execution platforms for the robot control language used in Fig. 2.2: Thymio (left), Lego Mindstorms NXT (right)

to a new active mode, *even when* a computation is active in another mode. Statements like the following describe the semantics of `robot`:

- *Reactions are only active if their mode is active.*
- *If a mode has a successor mode, then the control switches to it immediately after all actions have been executed.*

`Robot` is implemented in our online repository, in the project robot/ and related projects with the same prefix. We should now consider how the syntax and semantics of this language are implemented.

2.4 Building a Language

How do we get the robot control language presented in Fig. 2.2 to execute on a piece of real hardware, for instance on one of the robots shown in Fig. 2.3? The language implementation is split into five coarse aspects:

- *Concrete Syntax:* How does a language look to users? What do the users write or draw?
- *Abstract Syntax:* How are the models or programs of the language represented in the memory of a computer? What do the language designers use to implement the language?
- *Static Semantics:* What models or programs written in the language are legal? What models are erroneous (e.g., do not make sense)?
- *Dynamic Semantics:* An interpreter, a code generator, or a visualizer that gives computational meaning to the language.
- *Design Environment:* The tools for creating models and programs in the language (an IDE).

In the remainder of this chapter we demonstrate these key components using the `robot` example. For each of the five aspects we discuss a definition, an example, a way to specify or implement it, and the existing tool support.

[7] See also Chapter 14, especially Section 14.3, in the book of Matarić [38]

```
         Mode  →  '->'? Id  '{' ( Action | Reaction | Mode )*  '}'
                    ( '->'  Id )?
     Reaction  →  'on'  Event  '->'  Id
       Action  →  ( AcDock | AcMove | AcTurn )
                    ( 'for' AExpr 's' | 'at' 'speed' AExpr )?
       AcDock  →  'return'  'to'  'base'
       AcTurn  →  'turn' ( 'right' | 'left')? ( 'by' AExpr )?
       AcMove  →  'move' ( 'forward' | 'backward')
        AExpr  →  MinusMultExpr | PlusMultExpr
 PlusMultExpr  →  '+'? MultExpr ( ( '+' | '-' ) MultExpr )*
MinusMultExpr  →  '-' MultExpr ( ( '+' | '-' ) MultExpr )*
     MultExpr  →  Atomic ( ( '*' | '/' ) Atomic )*
       Atomic  →  RndI | INT | '(' AExpr ')'
         RndI  →  'random' ( '(' AExpr ',' AExpr ')' )?
        Event  →  'obstacle' | 'clap'
```

Figure 2.4: *A context-free grammar defining the syntax of the mobile robot control language* `robot`

Concrete Syntax

The concrete syntax of the language is the user interface of the language. This is what language users write or otherwise create. For textual languages, the concrete syntax is what is created in text editors and saved as files of characters. Figure 2.2 presents the random walk model in concrete syntax.

Specification. Any reader new to the topic surely appreciates how non-obvious it is to build tools that work with concrete syntax. Automatically extracting structure and meaning from a flat sequence of characters in a file requires non-trivial analysis. Fortunately, by now, this problem is extremely well understood, especially for simple languages like most DSLs. The concrete syntax for textual programming and modeling languages is typically defined using context-free grammars. An example for `robot` is found in Fig. 2.4. You can see there that a mode is written by starting with an optional arrow symbol (`'->'?`), followed by an identifier of the mode (Id), further followed by a list of actions, reactions, and modes enclosed in braces, and possibly followed by an identifier of the continuation mode. Such a specification is sufficient to automatically generate a parser, which will extract the core structure of the model from a text file and present it to computer tools as a data structure. We will explain the details in Chapter 4.

Tools. Context-free grammars are interpreted by automatic tools known as *parser generators*. A parser generator can automatically synthesize a *parser*—a front-end for your language tool that builds data structures out of textual input. A parser generated from a context-free grammar detects syntax violation errors, such as unmatched braces, missing keywords, lack of punctuation, etc. For `robot` it will, for instance, enforce that there has to be exactly one top-level mode, in which all the other modes are nested.

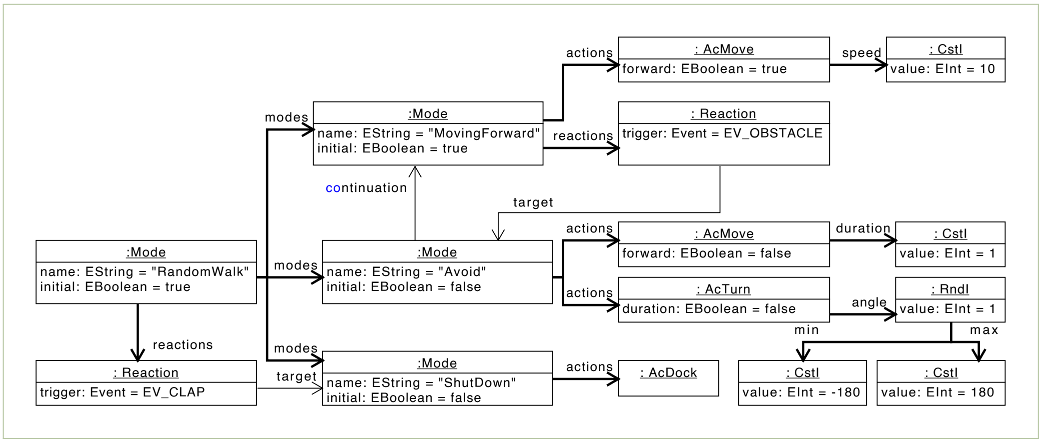


Figure 2.5: The abstract syntax of the random walk model. The bold lines indicate the tree structure

Abstract Syntax

The abstract syntax is a representation of a model or program inside computer memory. This is the representation seen by software processing the language—a compiler, a code generator, an interpreter, or an analyzer. The representation as a string of characters is unwieldy, as it does not capture the structure of the model/program well. Instead, the abstract syntax is represented as a tree of objects with cross references—an *abstract-syntax tree*, AST for short. An example of an abstract syntax for the model of the random walking robot (Fig. 2.2) is shown in Fig. 2.5 using the syntax of UML *instance specifications*. Each box represents an in-memory object capturing an element of the original model. The tree is rooted in a node representing a mode of the random walk. Follow the bold lines to see the tree structure clearly. The root mode contains three sub-modes, which further contain the actions and reactions. Try to establish an approximate correspondence of nesting in this diagram with the syntactic nesting in Fig. 2.2.

Specification. Since abstract-syntax trees are data structures, they are defined using types. Presently, two ways of specification are commonly accepted: class diagrams (classes in object-oriented languages) and algebraic data types (in functional programming languages). The types defining the abstract syntax are often referred to as a *domain model* or a *meta-model*. We shall discuss both ways of specification in Chapter 3.

Tools. In the implementation of DSLs, the abstract syntax is a *pivotal structure*: most language tools (parsers, importers, validators, converters, code generators, interpreters, visualizers) either produce or consume abstract syntax. This means that a good definition of an abstract syntax will allow you to separately develop and test various tool chain components, facilitating the parallelization of work, and its distribution among team members.

```
-> RandomWalkBroken {
  on clap -> ShutDown
  on clap -> Avoid

  MovingForward  {
    move forward at speed 10
    on obstacle -> Avoid
  }

  Avoid {
    move backward for 1 s
    turn by random (-180,180)
  } -> MovingForward

  ShutDown { return to base }
}
```

source: robot/test-files/random-walk-broken.robot

Constraint: *All reactions in the same mode should have distinct trigger events.*

```
inv[Mode] { self =>
      val triggers = self.getReactions.map { _.getTrigger }
      triggers.toSet.size == triggers.size }
```

Constraint: *A mode either has no sub-modes or it has an initial sub-mode.*

```
inv[Mode] { self =>
    (!self.getModes.isEmpty) implies
      (self.getModes.exists {_.isInitial}) }
```

Figure 2.6: *An incorrect model of a random walk robot controller, violating the static semantics rules in the bottom of the figure (presented in Scala).*

Static Semantics

The syntax of a language defines which models in the language are correct. Still, just like for spoken languages, syntactic correctness does not guarantee that a model makes sense. Consider the following sentence:

> *A context-free professor conjugates a well-typed glass of higher-order students.*

For most English speakers, the sentence will not appear correct, barring some poetic or psychedelic interpretations. This is despite the fact that it follows all basic grammar rules. The problem is that it violates commonly agreed ways to link words in a meaningful manner.

A similar problem arises for computer languages, where not all syntactically correct programs make sense. The *static semantics* eliminates many incorrect models and programs. It is concerned with aspects such as resolving name accesses (whether referred-to elements exist), ensuring that expressions are correctly typed (whether added elements are numbers), and so on. In the `robot` language, we may require that there is at most one reaction rule for each event in a mode, so that reactions do not compete with each other, or that any complex mode (mode with nested sub-modes) has an initial sub-mode. The model of a randomly moving robot of Fig. 2.2 satisfies both these rules, but the robot model in the top part of Fig. 2.6 violates both.

Specification. Typically, static semantics are specified by means of implementing a name analysis, type checking, and a number of validity constraints. Definitions of static semantics are much less standardized than definitions of syntax, although many theories and frameworks exist. Example constraints for the mobile robot control language are shown in the bottom of Fig. 2.6. More complex and sophisticated static checks can be considered, although most designers would limit themselves to properties that can be checked efficiently, to ensure good usability of the language tools. We discuss definitions of static semantics extensively in Chapters 5 and 6.

Tools. Some aspects of static semantics can be handled by language workbenches. For instance, Xtext has generic support for name analysis. Other tools use dedicated DSLs for writing semantics as models, which can then be automatically processed to enforce them. Examples include XSemantics [7], JetBrain's Meta-Programming System,[8] NaBL2[9] a part of the Spoofax workbench [28], and PLT Redex in the Scheme community [22]. The Object Constraint Language (OCL) [45] provides a standardized formalism, with several available implementations for specifying first-order constraints. It was originally created, among others, to give static semantics to languages, UML in the first place. Despite so many frameworks being available, we promote implementing static semantics in GPLs, in contrast to parsing concrete syntax, where manual implementations have gone out of fashion long ago. The proliferation of functional-programming constructs in mainstream GPLs allows concise and clean static semantics checks to be specified without specialized languages (Chapters 5 and 6).

Dynamic Semantics

Dynamic semantics is typically realized either by *interpreting* (e.g., executing, visualizing, or calculating) the models, or by *translating* to other languages, for which the meaning is known. Typically, it can be defined in multiple ways for the same language. We can give the meaning to `robot` by translating the models to a language executable on the target robotic hardware,[10] or by interpreting the models directly on a robot controller running a suitable robotics framework. One can also define the dynamic semantics abstractly, using a mathematical formalism. For `robot`, a suitable meaning would be a set of execution traces, listing modes, actions, and events. An example trace is: `RandomWalk`, `MovingForward`, `move forward at speed 10`, `obstacle`, `Avoid`, `move backward for 1 s`, `turn by 30`, `MovingForward`, `move forward at speed 10`, `clap`, `ShutDown`, `return to base`. The set of all execution traces would define the language formally and abstractly.

Figure 2.7 shows a fragment of an interpreter for `robot`, implemented in Scala, on top of Robot Operating System (ROS).[11] We do not expect

[8] https://www.jetbrains.com/help/mps/typesystem.html, retrieved 2022/08

[9] http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/index.html, retrieved 2022/08

[10] For example, Aseba script on a Thymio robot (http://wiki.thymio.org/en:asebalanguage, retrieved 2022/08), or URScript on a Universal Robot's arm [57].

[11] http://www.ros.org, retrieved 2022/08

```
 1 class Interpreter (root: Mode) extends NodeMain:

 3   var lock: Lock = ReentrantLock ()

 5   override def getDefaultNodeName (): GraphName =
 6     GraphName.of ("dsldesign/robot/scala/interpreter")

 8   override def onStart (cn: ConnectedNode): Unit =
 9     Thread.sleep (1000)
10     var state = State (dsldesign.robot.scala.Thymio (cn),
11                   Map[Event, Reaction] (), root)
12     var listener = new MessageListener[LaserScan] {
13       override def onNewMessage (msg: LaserScan): Unit =
14         // obstacle event
15         if msg.getIntensities.sum > 0.09 && lock.tryLock then
16           try state = state.processEvent (EV_OBSTACLE)
17           finally lock.unlock

19     if lock.tryLock then
20       try
21         state.thymio.getProximityTopic.addMessageListener (listener)
22         state = state.activate
23       finally lock.unlock
24   ...
```

source: robot.scala/src/main/scala/dsldesign/robot/scala/interpreter/Interpreter.scala

Figure 2.7: *An example of dynamic semantics implementation: the core part of the interpreter for* `robot`*, the mobile robot control language*

you to study how this is implemented. Instead, focus on the semantic gap between Fig. 2.7 (robot program interpreter) and Fig. 2.2 (robot program). In the interpreter, notice the concepts such as locks (line 3), threads (line 9), listeners and callbacks (lines 12–13), and exception handling (lines 16–17). All these concepts are necessary to implement the desired behavior in ROS, but they are absent in `robot` models. This clearly illustrates the nature of DSLs: a complex interpreter hides a large gap between the low-level language implementation and the input model, or between the *problem space* and the *solution space*. This gap is beneficial for the users of the language, who no longer have to worry about convoluted implementation concepts.

An additional function of dynamic semantics is to detect any remaining errors in models, introducing "last-minute" runtime validity checks—those that are necessary, but could not have been performed statically. It is well agreed between language experts that every non-trivial question about a program is undecidable.[12] Thus, static semantics can only guarantee well-formedness of models and programs to a limited extent. For a GPL, a property that is difficult to guarantee statically is the lack of divisions by zero. An example for a DSL could be whether there exists an instance satisfying all constraints in the diagram, or whether unsafe states are unreachable. If we need to enforce properties that are not checked statically, we insert them into dynamic semantics as runtime checks.

Specification. Dynamic semantics are usually implemented by building either an interpreter or a translator (a code generator). The differences, ad-

[12] Rice's theorem gives a formal reason [26, 48].

vantages, and disadvantages of various strategies are discussed in Chapter 9, where we build a code generator, and another interpreter, for `robot`.

Admittedly, by following the compiler terminology, we misuse the term *dynamic* semantics for DSLs. Unlike for GPLs, the semantics of many DSLs is not dynamic at all—they are not executable. For instance, DSLs for modeling structures (class diagrams), for modeling software configurations (feature models, see Sect. 11.4), or for modeling styling visualizations (CSS) are not directly operational. A CSS style sheet mostly describes how things "look" not how things "behave." In such cases, we basically mean that the dynamic semantics is the implementation of the back-end of the language-processing tools, for instance a code generator for class diagrams, a renderer for CSS, or an interactive configurator for feature models.

Tools. Some of the same tools that can be used to specify static semantics can also allow us to define dynamic semantics by defining operational *reduction rules* in a specialized DSL (see Chapters 7 and 8). The reduction rules define an evolution of the system by specifying how an expression specifying a system's state evolves over time through rewriting, analogously to how rewriting can be used to calculate a mathematical expression. Specialized transformation languages are well suited for this purpose. However, it is still most common to implement the dynamic semantics directly in a GPL. Especially modern functional programming languages such as Haskell, F#, and Scala lend themselves very well to this task. Their predecessor, the ML language (later known as Standard ML or SML) was in fact designed with meta-programming as a primary use case. The early application of ML was the implementation of the theorem proving system LCF [24], where syntax trees of formulae had to be rewritten in proof rules. This task is very similar to writing an interpreter. We discuss the implementations of language back-ends in various languages in Chapters 7 to 9.

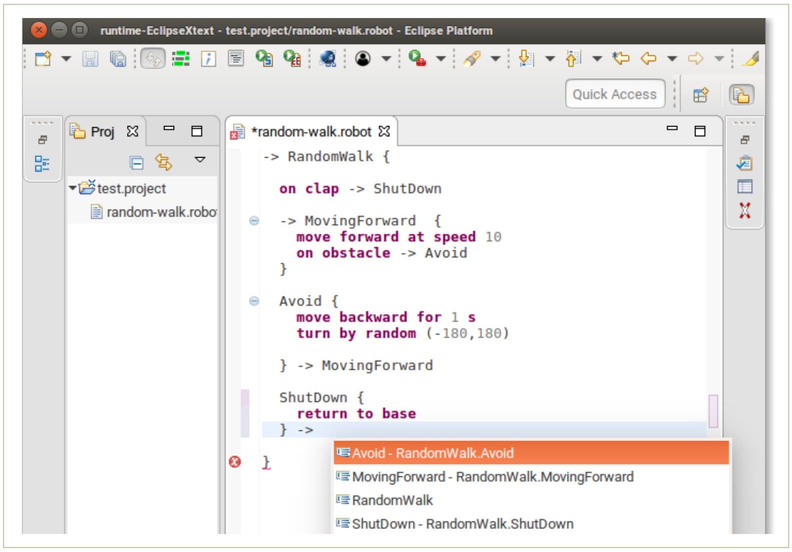
Design Environment

A modern programmer expects rich editing environments with syntax and error highlighting, code completion, and name resolution; integrated with test infrastructure, and project navigation. A screenshot of a generated editor for the mobile robot control language is presented in Fig. 2.8 (produced using the Xtext framework[13] described in Sect. 4.4).

Specification. The tools used to generate development environments depend largely on the specification of syntax and static semantics. The editor shown in Fig. 2.8 has been generated from an Xtext language model, defining the abstract and concrete syntax, and the static semantics. Specifying other tools (testers, analyzers, debuggers) often requires direct implementations, following similar patterns to those for interpreters and generators.

Tools. *Language workbenches* integrate all language implementation components discussed above, allowing an integrated development environment,

[13] http://www.eclipse.org/Xtext/, retrieved 2022/08


runtime-EclipseXtext - test.project/random-walk.robot - Eclipse Platform
Quick Access
Proj
test.project
random-walk.robo
*random-walk.robot

```
-> RandomWalk {

  on clap -> ShutDown

  -> MovingForward  {
    move forward at speed 10
    on obstacle -> Avoid
  }

  Avoid {
    move backward for 1 s
    turn by random (-180,180)

  } -> MovingForward

  ShutDown {
    return to base
  } ->

}
```

```
Avoid - RandomWalk.Avoid
MovingForward - RandomWalk.MovingForward
RandomWalk
ShutDown - RandomWalk.ShutDown
```

Figure 2.8: A feature-rich editor generated by Xtext for `robot`*, our mobile robot control language*

an IDE, to be generated or otherwise created [20]. A language workbench typically includes a parser generator for handling concrete syntax, along with some facilities for specifying static and dynamic semantics. Workbenches generate editors that combine all the language definition components to provide semantically aware editing: the editor can resolve names, complete references, parse, build an AST, check for validity, and possibly also execute the model. Some language workbenches can automatically create web-based editors, which you can deploy as part of web applications.

An interesting recent addition to this technology is the Language Server Protocol (LSP),[14] which allows generic support of rich IDE functionality in any editor and language for which this protocol is implemented [5]. This means that the cost of creating a rich editing experience is reduced dramatically. Once you use a language workbench that can automatically create an LSP server for your language, you obtain a rich experience in any editor implementing an LSP client (which presently includes all major editors).

We have now briefly surveyed all major components of a language implementation. Table 2.1 summarizes briefly the above developments. In the first column, we list the language design components discussed above, along with references to chapters that discuss them in detail. For each language component we state the purpose, the way to specify/implement it, and the tools that work with this component.

2.5 Testing Language Implementations

Testing is by far the most popular and (so far) the most effective way to assure the quality of software components. When moving regular software

[14] https://langserver.org/, retrieved 2022/08

Language Component	**Purpose**	**Specification Examples**	**Example Tools**
Concrete syntax Chapter 4	Writing and reading interface for the language: language users write and read programs in concrete syntax.	Regular expressions and context-free grammars.	Parser generators and parsers.
Abstract syntax Chapter 3	An in-memory representation of models and programs as structures in a programming language; a pivotal structure used by the front-end and back-end of the language infrastructure. This is what the language designer uses to implement the language.	Algebraic data types or meta-models.	Produced by parsers, consumed by transformations. Visualized as diagrams or trees for debugging in IDEs.
Static semantics Chapters 5 and 6	Defining valid/invalid models; enforcing well-typedness/constraints impossible/hard to express with grammars and meta-models/ADTs.	First-order constraints, inductive type-system rules, scoping rules.	Advanced frameworks exist, but still mostly implemented manually in practice.
Dynamic semantics Chapters 7 to 9	Define meaning of programs and models; realize the actual purpose of the models.	Code generator or interpreter implemented in a transformation language or in a high-level functional language.	Advanced frameworks exist, but still most languages are implemented manually in practice.
Design environment Chapter 4	Supporting users in creating domain-specific models. The modern editor for your specialized language.	Uses specifications for the other components.	Language workbenches generate high-quality comfortable editors.

Table 2.1: Overview of language infrastructure components.

projects to MDSE, we have to recognize that testing is also important for DSL infrastructure. Large parts of the logic of our projects will be embedded in language definitions, in interpreters and generators.

Consider the implementation of `robot`. To test whether the concrete syntax is sufficiently well specified, we need a good collection of models of robots that should parse (i.e., our generated parser recognizes them successfully), and also a collection of models of robots that contain syntactic errors and should not parse. Similarly, for the static semantics, we need to gather cases of models that should and should not produce static checking errors. Test cases for syntax and for static semantics are usually created at design time, or ahead of design time, and are later extended with regression test cases, as problems are discovered during development and usage of the language. For instance, the model in Fig. 2.2 could be used as a test case for both syntax and static semantics, while the model in Fig. 2.6 could be used as a negative test case for static semantics, and a positive test case for the concrete syntax (parsing).

It is considerably harder to test the implementations of the dynamic semantics. For manual testing, we can create diverse `robot` models, run the interpreter for each of them, and check whether the system behavior is consistent with the model. This can be improved slightly, by replacing the physical robot with a simulator and using the so-called *model-in-the-loop* testing (MIL). But how can we test the dynamic semantics automatically? Without automatic tests, we can forget about test-driven development and continuous integration. This would lead to a drop of quality in our project, while our goal is exactly the opposite. One possibility is to create an execution harness for the interpreters of models. Our test cases then become triples: a model of a robot, a sequence of inputs, and a sequence of expected actions.

In this book, we discuss testing patterns for each of the language aspects in the corresponding chapter. We define a suitable notion of test cases, discuss what it means for a test to pass or fail (oracles), elaborate on possible stop criteria for testing (notions of coverage), as well as testing architectures and patterns for language implementations.

Further Reading

While many of the technologies presented here can be used for designing and building general-purpose languages (GPLs), we focus on low-expressiveness languages (DSLs) and side-step the specialized advanced concepts that one needs for realizing compiled or interpreted GPLs, such as advanced type systems, optimization, or generation of machine code. For such topics, we refer to literature on compilers and programming languages [1, 42, 50].

Fowler and Parsons [23] give a good, if somewhat traditional, coverage of design issues and implementation techniques for DSLs. They also include a good initial introduction to internal DSLs, with a very interesting collection of implementation patterns using various mechanisms of the host language. There is also a book about Microsoft DSL Tools [16], but the tool does not seem to be maintained anymore. The MetaEdit+ tool from MetaCase has an associated book that shows a good set of examples and principles to follow, especially for graphical DSLs [30].

Bettini [6] gives a very pragmatic, even hands-on, course on development of textual DSLs with the Xtext framework. Since this is the same framework as used in this book, Bettini's volume is a very convenient companion. While we focus more on general aspects of language design, and present the methods as far as possible in a tool-independent manner, Bettini explains directly how to work with Xtext.

The concept of a language workbench is attributed to Martin Fowler.[15] Language workbenches are advocated in detail by Völter [60]. A good overview of the features offered by modern workbenches is surveyed by Erdweg et al. [20]. The technology, under various disguises and at various levels of maturity, has existed since the 1980s. Most workbenches were originally designed to facilitate the creation of general-purpose programming languages, and were adopted over time for designing DSLs. Early workbenches for textual language included: SEM [56], MetaPlex [12], Metaview [55], Centaur [10], QuickSpec [37], MetaEdit [53], ASF+SDF Meta-Environment [32], Gem-Mex/Montages [2], LRC [35], and LISA [41]. Contemporary workbenches for textual languages are JastAdd [25, 54], Rascal [33], Spoofax [28], Melange [19], Xtext [21, 6], MontiCore [34], and Neverlang [58], just to name a few.

The workbenches for graphical syntax include Sirius, mentioned above, MetaEdit+ [29], and the open-source tools building upon the Eclipse Modeling Framework: the Graphical Modeling Framework and Graphiti, which are both part of Eclipse's Graphical Modeling Project.[16] Two less known graphical workbenches are DOME [11] and GME [36].

[15] https://martinfowler.com/articles/languageWorkbench.html, retrieved 2022/08

[16] https://www.eclipse.org/modeling/gmp, retrieved 2022/08

Projectional editing, also known as structured or syntax-directed editing, goes back to the 1980s, with tools such as the Incremental Programming Environment [40], GANDALF [44], and the Synthesizer Generator [47]. Projectional editing became popular with the Intentional Programming paradigm, which puts language composition at the core of software engineering [51, 17]. Today, Jetbrains Meta Programming System (MPS)[17] and Intentional's Domain Workbench [52, 14] are the most comprehensive projectional language workbenches.

Schauss et al. [49] illustrate many technologies for construction of DSLs. The paper is accompanied by a code repository[18] containing examples of DSL implementations created with several workbenches (including Eclipse Modeling Framework, Java/ANTLR, Rascal, JetBrains MPS, and Spoofax), as well as several embedded DSLs (Scala, Rascal, and Racket).

In this chapter, we sketched an implementation of a simple external DSL for robot control. Peterson, Hudak, and Elliott [46] demonstrate an internal DSL (an API-like language), for a similar purpose but using an entirely different implementation pattern. The distinction between internal and external DSLs will be made clearer in later chapters of the book. Robotics is not an accidental choice for our example. Given the complexity of robotics systems, and a range of well-defined tasks and activities in robotics, DSLs are often a natural choice to formalize designs. Not surprisingly, DSL proposals proliferate in this space. Already low-level robotics frameworks (such as ROS) use many DSLs for describing packages, interfaces, builds, deployments, hardware, scene, etc. Many more DSLs are built at a higher level of abstraction, aiming for more complex aspects of robots such as reasoning, planning, kinematics, and system architecture. See Nordmann et al. [43] and its accompanying website[19] for a recent list of more than hundred papers describing robotics DSLs.

Additional Exercises

Exercise 2.1. **a)** Revisit Example 5 on p. 29. Using two different colors, highlight all sentences (or sentence fragments) specifying syntax (respectively semantics) of the `robot` language. Observe that in informal language descriptions syntax and semantics are often mixed. **b)** Find a short informal description (or a fragment of description) of a computer language relevant for you. Select an interesting fragment, and repeat the highlighting exercise on this fragment.

Exercise 2.2. Identify an educational platform for robotics of your choice. A typical educational platform will offer several APIs in GPLs and some DSLs at various levels of sophistication, to cater for users programming the robots at different stages of education. Pick two of these interfaces (either APIs and/or DSLs) from whatever is available, and analyze them. Study tutorials briefly, and read through some code examples. When discussing the properties of the interfaces, try to contrast the two choices you made.

Attempt to answer the following questions for DSLs (if any): **a)** Who is the target user? **b)** What use cases are supported by the language? **c)** What is the expressiveness of the language? Is it in any way limited? Are any robotics-specific

[17] http://www.jetbrains.com/mps/, retrieved 2022/08
[18] https://softlang.github.io/metalib/, retrieved 2022/08
[19] http://corlab.github.io/dslzoo/index.html, retrieved 2022/08

tasks easier in this language than in general-purpose programming languages? Are there any general programming tasks that are difficult to perform in this DSL?

Answer the following questions for selected APIs in GPLs (if any): **d)** Who are the expected target users for this API? For what use cases? **e)** Are there any API elements that are not directly pertinent to robotics tasks? Would it be possible to eliminate any of them using a DSL? **f)** Does using the API involve a lot of boiler plate code? Is creation of this code likely possible to automate? **g)** Your opinion: Is the API a suitable target to use implementing an interpreter for a DSL, or is it a good target for code generation? If there are several APIs available, perhaps consider which one would be the easiest to use as a back-end for the DSL.

Exercise 2.3. For the robotics framework studied in Exercise 2.2 investigate what testing and quality assurance support is provided by the vendor, or the framework's open source ecosystem.

Exercise 2.4. Chef[20] is a deployment and configuration management language. It started as an internal DSL implemented in Ruby and has grown out into a proper external DSL. Discuss Chef based on what you learned about models and DSLs in this chapter, specifically: What is the domain described by models in Chef? What information is present, what is abstracted away, hidden? What is the style of the syntax of this language? What tasks are automated thanks to Chef? From where does the Chef infrastructure take information to execute simplistic models? Browsing through the slides of a Webinar on Chef[21] should suffice for this discussion.

Chef itself is of no particular importance for the rest of this book. You can replace Chef with any other DSL. For example, if you are interested in robotics, visit the *Robotics DSL Zoo* [43], pick one of the languages that attracts your attention, and execute the above discussion for this language. This is an open exercise with no perfect answer. It is meant to help you explore the concepts.

Exercise 2.5. Imagine a hypothetical configuration application, where a number of parameters need to be configured, to satisfy an input model. The configurator is equipped with a plugin mechanism, so that it can load .jar files containing more sophisticated calculations that are made available in the model constraints.

The configuration tool can report various error messages for an input model. For each of them decide which part of the DSL implementation is reporting the error. Justify your answers briefly. Some possible answers include the interpreter, the type system, regression tests, the code generator, constraints, the parser, etc...

a) `Could not find the external dependency 'FunctionalCalculations.jar'. No such file or directory.`

b) `line 213: Expected keyword 'parameter' instead of EOL`

c) `Parameter group 'Engines' depends on itself`

d) `line 196: Expected an Integer value instead of String`

e) `The enumeration type 'color' should have distinct values. Value 'pink' is repeated in lines 400 and 404.`

[20] https://learn.chef.io/, retrieved 2022/08

[21] http://www.slideshare.net/chef-software/overview-of-chef-fundamentals-webinar-series-part-1, retrieved 2022/08

Exercise 2.6. Discuss informally what testing (quality assurance) process you would carry out to ensure that the grammar presented in Fig. 2.4 captures the right models in the robot control language—admits the models of interest as legal, and rules out the models that are syntactically incorrect.

Exercise 2.7. Informally discuss the selection of test cases for the two constraints in Fig. 2.6. How many and what test cases would you select for each of the constraints? If you have a system where there are many other constraints, and you are time limited in testing, what would be the most important test cases?

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Prentice Hall, 2006 (cit. p. 40).

[2] Matthias Anlauff, Philipp W. Kutter, and Alfonso Pierantonio. "Tool support for language design and prototyping with montages". In: *International Conference on Compiler Construction*. Springer. 1999 (cit. p. 40).

[3] John W. Backus et al. "Revised report on the algorithmic language ALGOL 60". In: *Commun. ACM* 6.1 (Jan. 1963). Ed. by P. Naur, pp. 1–17 (cit. p. 25).

[4] John W. Backus et al. "The FORTRAN automatic coding system". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). Los Angeles, California: ACM, 1957 (cit. p. 25).

[5] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. "Editing support for software languages: Implementation practices in language server protocols". In: *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2022 (cit. p. 38).

[6] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt, 2013 (cit. p. 40).

[7] Lorenzo Bettini. "Implementing Java-like languages in Xtext with Xsemantics". In: *Symposium on Applied Computing*. ACM. 2013 (cit. p. 35).

[8] Jean Bézivin. "On the unification power of models". In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188 (cit. pp. 25, 29).

[9] Andrew D. Booth and Kathleen H. V. Britten. *Coding for the A.R.C.* 1947 (cit. p. 25).

[10] Patrick Borras, Dominique Clément, Th Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. "Centaur: the system". In: *ACM SIGPLAN Notices* 24.2 (1988), pp. 14–24 (cit. p. 40).

[11] Honeywell Technology Center. *DOME guide*. 1999 (cit. p. 40).

[12] Minder Chen and Jay F. Nunamaker. "Metaplex: an integrated environment for organization and information system development". In: *International Conference on Information Systems. Proceedings*. 1989 (cit. p. 40).

[13] Noam Chomsky. *Syntactic Structures*. Mouton & Co., 1957 (cit. p. 29).

[14] Magnus Christerson and Henk Kolk. *Domain Expert DSLs*. A talk at QCon London 2009. 2009. URL: http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk (cit. p. 41).

[15] COBOL-1961. *Report to conference on data systems languages*. Tech. rep. US Dept. of Defense, 1961 (cit. p. 25).

[16] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007 (cit. p. 40).

[17] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000 (cit. pp. 27, 41).

[18] Ole-Johan Dahl and Kristen Nygaard. “Class and subclass declarations”. In: *IFIP TC2 Conference on Simulation Programming Languages*. 1967 (cit. p. 26).

[19] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. “Melange: a meta-language for modular and reusable development of DSLs”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 2015 (cit. p. 40).

[20] Sebastian Erdweg et al. “The state of the art in language workbenches”. In: *SLE*. 2013 (cit. pp. 27, 38, 40).

[21] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010*. 2010 (cit. p. 40).

[22] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009 (cit. p. 35).

[23] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley, 2011 (cit. pp. 26, 40).

[24] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Vol. 78. Lecture Notes in Computer Science. Springer, 1979 (cit. p. 37).

[25] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58 (cit. p. 40).

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001 (cit. p. 36).

[27] Paul Hudak. “Building domain-specific embedded languages”. In: *ACM Comput. Surv.* 28.4es (1996), p. 196 (cit. p. 26).

[28] Lennart C.L. Kats and Eelco Visser. “The Spoofax language workbench”. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. SPLASH/OOPSLA 2010. ACM, 2010 (cit. pp. 35, 40).

[29] Steven Kelly, Kalle Lyytinen, and Matti Rossi. “MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 1996 (cit. p. 40).

[30] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008 (cit. p. 40).

[31] Anneke G. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2009 (cit. p. 29).

[32] Paul Klint. “A meta-environment for generating programming environments”. In: *ACM Trans. on Soft. Eng. and Method. (TOSEM)* 2.2 (1993), pp. 176–201 (cit. p. 40).

[33] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “Rascal: a domain specific language for source code analysis and manipulation”. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2009 (cit. p. 40).

[34] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: A framework for compositional development of domain specific languages”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.5 (2010), pp. 353–372 (cit. p. 40).

[35] Matthijs Kuiper and João Saraiva. “Lrc—a generator for incremental language-oriented tools”. In: *International Conference on Compiler Construction*. Springer. 1998 (cit. p. 40).

[36] Akos Ledeczi et al. “The generic modeling environment”. In: *Workshop on Intelligent Signal Processing, Budapest, Hungary*. Vol. 17. 2001 (cit. p. 40).

[37] Meta Systems Ltd. *Quickspec reference guide*. 1989 (cit. p. 40).

[38] Maja J. Matarić. *The Robotics Primer*. MIT Press, 2007 (cit. p. 31).

[39] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, Part I”. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195 (cit. p. 25).

[40] Raul Medina-Mora and Peter H. Feiler. “An incremental programming environment”. In: *IEEE Trans. Softw. Eng.* 7.5 (Sept. 1981), pp. 472–482 (cit. p. 41).

[41] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. “LISA: An interactive environment for programming language development”. In: *International Conference on Compiler Construction*. Springer. 2002 (cit. p. 40).

[42] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, 2011 (cit. p. 40).

[43] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. “A survey on domain-specific modeling and languages in robotics”. In: *Journal of Software Engineering in Robotics (JOSER)* 7.1 (2016), pp. 75–99 (cit. pp. 41, 42).

[44] David Notkin. “The GANDALF project”. In: *J. Syst. Softw.* 5.2 (May 1985) (cit. p. 41).

[45] Object Management Group. *OCL Specification version 2.2*. http://www.omg.org/spec/OCL/2.2/. 2010 (cit. p. 35).

[46] John Peterson, Paul Hudak, and Conal Elliott. “Lambda in motion: Controlling robots with Haskell”. In: *Practical Aspects of Declarative Languages (PADL)*. Ed. by Gopal Gupta. Vol. 1551. 1999 (cit. p. 41).

[47] Thomas Reps and Tim Teitelbaum. “The Synthesizer Generator”. In: *ACM SIGPLAN Notices* 19.5 (1984), pp. 42–48 (cit. p. 41).

[48] Henry G. Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. p. 36).

[49] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. “A chrestomathy of DSL implementations”. In: *10th International Conference on Software Language Engineering (SLE)*. 2017 (cit. p. 41).

[50] Peter Sestoft. *Programming Language Concepts*. Springer Science & Business Media, 2012 (cit. p. 40).

[51] Charles Simonyi. “The death of computer languages, the birth of intentional programming”. In: *Proc. NATO Science Committee Conference*. 1995 (cit. p. 41).

[52] Charles Simonyi, Magnus Christerson, and Shane Clifford. “Intentional software”. In: *Proceedings of OOPSLA*. 2006 (cit. p. 41).

[53] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. “MetaEdit—a flexible graphical environment for methodology modelling”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 1991 (cit. p. 40).

[54] Emma Söderberg and Görel Hedin. “Building semantic editors using JastAdd: Tool demonstration”. In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. 2011 (cit. p. 40).

[55] Paul G. Sorenson, Jean-Paul Tremblay, and Andrew J. McAllister. “The Metaview system for many specification environments”. In: *IEEE Software* 5.2 (1988), pp. 30–38 (cit. p. 40).

[56] Daniel Teichroew, Petar Macasovic, Ernest Hershey, and Yuzo Yamamoto. “Application of the entity-relationship approach to information processing systems modeling”. In: *Entity-Relationship Approach to Systems Analysis and Design*. Ed. by P.P. Chen. North-Holland, 1980 (cit. p. 40).

[57] Universal Robots. *The URScript Programming Language. Version 3.1*. Jan. 2015 (cit. p. 35).

[58] Edoardo Vacchi and Walter Cazzola. “Neverlang: A framework for feature-oriented language development”. In: *Computer Languages, Systems & Structures* 43 (2015), pp. 1–40 (cit. p. 40).

[59] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. “Sirius: a rapid development of DSM graphical editor”. In: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE. 2014 (cit. p. 27).

[60] Markus Völter. *DSL Engineering. Designing, Implementing and Using Domain Specific Languages*. 2013. URL: http://www.dslbook.org (cit. pp. 29, 40).

[61] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. “Comparative analysis of DSM graphical editor frameworks: Graphiti vs. Sirius”. In: *Proceedings of the 23rd International Electrotechnical and Computer Science Conference (ERK’14)*. 2014 (cit. p. 27).

There are catalogues of catalogues.
There are poems about poems.
There are plays about actors played by actors.
Letters due to letters.
Words used to clarify words.

Wisława Szymborska [23]

3 Domain Analysis and Abstract Syntax

You want to design a DSL to boost software development, evolution, or customization in some domain. In the first step, you need to clarify what are the key relevant aspects of this domain, in a process known as *domain analysis and meta-modeling*. During the analysis, we identify the relevant concepts and relationships between them. During meta-modeling, we formalize this knowledge in a model, and iteratively refine it until the model precisely describes the *abstract syntax* of the DSL. It will define which models or programs we shall be able to write in your language.

We now discuss these steps in detail. The chapter includes design and analysis guidelines for meta-models, discusses them on a running example (a DSL of finite-state machines), touches upon several meta-modeling languages (meta-meta-models) with a focus on class diagrams, and then explains what instances (called models) of meta-models look like. Finally, we explain how *models*, *meta-models*, and *meta-meta-models* relate in the theoretical framework known as the *language-conformance hierarchy*.

3.1 What is Meta-Modeling?

Let us define the prime outcome of domain analysis and meta-modeling.

Definition 3.1. *A meta-model is a model that precisely defines the parts and rules needed to create valid models in a DSL [7].*

The parts refer to the domain concepts captured in the language, while the rules are any kind of constraints that prescribe the construction of valid models (i.e., instances) from those parts. Note that when we say *valid models*, we refer to their abstract syntax, which is independent of the actual notation (concrete syntax) of the language. While mappings need to be defined between the abstract syntax and any of the concrete syntaxes, a meta-model only determines the abstract syntax of a model. Whether the concrete syntax of a model is correct has to be assured by other means, such as a grammar for a textual DSL (Chapter 4).

Definition 3.2. Abstract syntax *is a representation of a program (model) in computer memory as a data structure, usually a tree or an instance of an object-oriented meta-model.*

We say that *meta-modeling* is the practice of modeling other modeling languages, and a meta-model is a model of a modeling language. The prefix "meta" comes from Greek, and today it means "self-referential." Meta-modeling is self-referential in the sense that it models modeling.

Practicing meta-modeling requires a meta-modeling language, often called a meta-meta-model. *A meta-modeling language is a modeling language containing concepts that allow us to conveniently describe the abstract syntax of other languages.* Especially in the context of MDSE we need expressive and precise meta-modeling languages, so that we can generate the infrastructure for DSLs automatically. Class diagrams are probably the most common meta-modeling language. However, other languages can also be used, such as feature models [10, 17], which are also precise, but less expressive than class diagrams. We discuss feature models extensively in Chapters 11 and 12. Other popular languages used for meta-modeling are XML Schema and other schema languages, as well as data types in most programming languages, in particular algebraic data types in functional languages (see also Sect. 7.1).

In this book, we work with two meta-modeling notations: class modeling and algebraic data types, as key representatives of two language design traditions. *Ecore* from the Eclipse Modeling Framework (EMF, see the box on p. 53) is a class-modeling language popular in the object-oriented programming community. It is less expressive and simpler than UML class diagrams. Many open-source tools exist around Ecore—they can be used to manipulate models and to implement languages. Algebraic data types (ADTs) are used to express the abstract syntax of languages by functional programmers. Most functional programming languages enjoy good support for processing language representations easily and efficiently.

Meta-modeling is useful not only for language modeling, but also for domain modeling and for model interchange between different tools [9], where meta-models are essentially the exchange formats. An example standard is the XML Metadata Interchange format XMI [19], which is defined as part of MOF.

3.2 Domain Analysis for Meta-Modeling

DSLs tend to be designed for reasonably mature and well-understood domains, to capitalize on the insights and experiences accumulated during years of engineering practice. Most often, a non-model-driven system, or even several ones, already exist in the area. Either you, your team, your customer, or other experts will be able to describe the key requirements for the new language. Similarly, existing examples of concepts of interest (cases, drawings, informal models, data entries, API usages) and documentation of prior practice are useful inputs to domain analysis and meta-modeling.

We demonstrate domain analysis with an example familiar to any computer science undergraduate: finite-state machines. The example is specifically selected so that we can sidestep the issues of missing knowledge about the domain and settle the basic terminology on familiar grounds.

Domain Models and Meta-Models

You are probably familiar with domain modeling—an activity often included in the early stages of software design in general, for example using UML diagrams. Domain modeling is a close relative of meta-modeling: the reasoning and abstractions used in creating both kinds of models are similar. Yet, they differ in their primary purpose and the level of formality and precision.

A *domain model* describes relevant concepts and their relationships in a particular domain. A domain model is typically created early in a software project. Sometimes, domain models are reverse-engineered when a project already exists and developers or domain experts want to create an overview of the relevant concepts, for instance before introducing any changes. Many project teams express domain models using class diagrams, but other languages, such as mind maps, feature models, state diagrams, or informal drawings, are also used. The main purpose of a domain model is to facilitate understanding and communication among persons involved in a project, including users, domain experts, developers, and architects.

A meta-model represents a domain *as a language*. A meta-model tends to be more formal than a domain model; it aims at precisely describing the possible instances (models or programs) of the language. A meta-model *models* a language. The emphasis on precision allows: (i) building MDSE tooling to generate language infrastructure, such as comfortable editors (with code-completion, error markers, and syntax highlighting), serializers, and deserializers, (ii) automatically checking that language instances conform to the meta-model, and (iii) implementing the semantics (e.g., an interpreter or a code generator) of the meta-model. Only secondarily, meta-models are also a unit of communication. Since both domain models and meta-models describe domain concepts, there is significant similarity between these concepts.

Example 6. We design a language `fsm` for describing sets of parallel *finite-state machines*. Computer science students will use this language to specify examples and solve exercises. They need to execute the models to interactively explore behaviors. Each state machine has a name and a number of named states. One state of each machine is singled out as an initial state. Transitions connect pairs of states: a source and a target state. Each transition is labeled by an input action and, optionally, by an output action.

This description is the input to our hypothetical domain analysis. In reality, it would have been extracted by interviewing stakeholders and studying available documents. Table 3.1 organizes the example description by the five questions discussed below. Study the table before proceeding.

Key questions and activities. During the domain analysis you should ask yourself and the subject matter experts the following five questions:

Q1: Purpose. *What is the purpose of the language? What are the use cases?* Concrete operational examples effectively guide the language design. Ask your users and experts what use cases are important and how they are realized today. Ask to prototype entirely new scenarios; ask how they imagine work with use cases not seen in existing systems or processes.

You will use the collected use cases to design a language that is as small as possible, narrowed down to a minimum set of concepts. Resist the urge to add things that are “nice to have.” Focusing on the required use cases lowers the development and adoption costs without hampering usability.

Q2: Stakeholders. *Who are the key stakeholders and the intended users of the language?*

A language for software developers or system administrators poses different requirements than a language for an electrical engineer or a fire-alarm installation consultant. Without considering the users, it is practically impossible to set the right level of abstraction. Bring the user personas in focus, to build the language on the terms and ideas that they are familiar with. Understand what are their organizational roles, and what background expertise they have [24]. Later, this will also help you to select a suitable concrete syntax.

Q3: Concepts. *What are the key domain concepts that users care about?*

Enumerate the concepts of importance, including physical, structural, logical, abstract, concrete, operational, and temporal concepts. This includes anything that is necessary to describe in order to build an unambiguous model for your use case. Do not limit yourself to static concepts that represent physical objects in the domain, such as an “engine” or an “engine controller.” It is equally important to also capture more transient concepts representing activities (e.g., a “fuel-injection policy”) and temporal properties (e.g., “rotation frequency” or a “weekly assignment rotation”). Many people have a natural tendency to focus on the static concepts and will forget to tell you about the transient ones, unless asked specifically.

A common mistake is to include concepts and relations that belong to the technical context of the DSL, but are not in the syntax of the language. For example, the interpreter for state machines in Example 6 is not a part of our language, but an associated tool. Therefore it does not belong to our meta-model. The meta-model only includes concepts that must be describable in the DSL, and not parts of the architecture, such as what tools and processes we will run. These are still listed in the use cases, though (Q1). The interpreter is important in the example, as we have to ensure that the models capture all the information needed for execution. However, the interpreter itself will not be modeled in the state machines and is not among the listed concepts in Tbl. 3.1.

Q4: Relations. *How are domain concepts related, and what are their relevant properties?*

Relations and properties organize and restrict your meta-model. For instance, “every engine needs an engine controller, but only certain controllers are suitable for hybrid engines.” The relations might not be static: “A student is enrolled at the university until she graduates.” The relations may also relate transient concepts: “fuel injection policy” is applied to an engine while in force, and not otherwise. Customarily, properties (or attributes) are relations to very simple concepts such as age or color, which do not require further elaboration.

Relations often emerge already in discussion of concepts (Q3). Do not try to artificially separate the discussion of concepts from the discussion of their relations in early design stages. Many relations can be seen as concepts and vice versa, so it is not useful to draw the distinctions sharply at this stage.

Q5: Examples. *What examples of language instances are available or can be prototyped?*

Using examples is a key technique, when eliciting requirements from domain experts. You should collect existing examples, ask subject matter experts to sketch new ones, and build some yourself to seek confirmation of your understanding.

Perhaps your customer is already using a notation or conventions expressing the subject of the DSL. In this case, an important objective for the DSL may be to formalize the existing notations, so that tools can be build and the automation of MDSE can be unleashed. Sometimes, you can define the language solely based on the existing notations and conventions.

Alternatively, the new language may be built to raise the level of abstraction of existing notations. For instance, a complicated and rich language (a GPL) can be replaced by a simpler intuitive and task-oriented language. Then, it is still valuable to understand existing notations by collecting examples, but it is necessary to build small examples of the new abstract language with the users in order to judge how well they are suited for the actual purpose.

These questions should not be answered in a sequential, waterfall-like process. We recommend to *perform the domain analysis iteratively* (indeed, the entire DSL design-and-implementation process should be iterative). Collect as little information as seems necessary, then build some examples and return to the subject matter experts for verification. Only collect new information if you are unable to support the use cases.

Exercise 3.1. In a group, pick a domain of interest, and perform the domain analysis following the above questions. If you do not know what to choose, consider modeling a format of a boarding pass for a flight, a course front page for a course management system such as Moodle, layout of furniture in a classroom, light scenarios for a classroom, or a deployment architecture of a simple web-based system. One person, with an idea how the language for the chosen domain should work, takes the role of a domain expert. The others play the language engineers. Build a table similar to Tbl. 3.1.

3.3 Meta-Modeling with Class Diagrams

How can we turn the knowledge collected in a domain analysis into a meta-model? We have to encode it in a formal language well suited for meta-modeling. Once we have a formal meta-model, we will implement our DSL using MDSE. For the state-machine example (Example 6), we

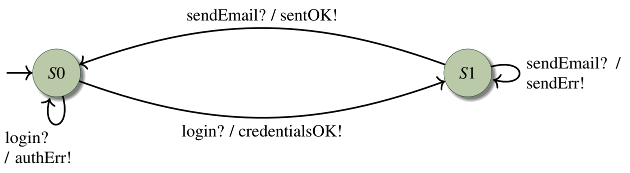
Q1: Purpose	To build examples of student exercises; To interact with examples using an interpreter, an interpreter will be needed.
Q2: Users	Computer science students learning automata theory (probably knowing the basics of a programming language); A professor, who can provide the examples and will ask the students to use the tool.
Q3: Concepts	Finite-state machines, several in parallel; States; Transitions;
Q4: Relations	*Properties:* states may be initial or end states, states and machines have names, transitions have input action labels, transitions have optional output labels; *Relations:* machines *own* states, transitions *connect* source and target states.
Q5: Examples	The professor whose students are supposed to use the tool provided us with the following example of a model in concrete graphical syntax:  sendEmail? / sentOK! S0 S1 sendEmail? / sendErr! login? / credentialsOK! login? / authErr!

Table 3.1: Knowledge collected in a hypothetical domain analysis process for the state-machine example

will use MDSE not only for generating code from state machines, but also when designing and implementing the state-machine language itself. This has an additional advantage, that you, the designer of the DSL, use the same paradigm as the users of your DSL. This makes you a more empathic designer, able to understand users' requirements better.

In this section, we use a minimalistic subset of class diagrams called Ecore to build meta-models (see the side box "Ecore, MOF, and Meta-Modeling"). If you lack experience with class diagrams, please study the appendix "Class Modeling" on our book website (http://dsl.design) before reading further.

Object-oriented analysis and design. When meta-modeling with class diagrams, we follow the principles of object-oriented analysis and design. We name classes after concepts and use associations to represent relations. Containment associations represent *part-of relations*; most other relations are specified as regular associations with suitable role names. Generalization (also known as inheritance) captures *kind-of relations* between concepts.

Example 7. Figure 3.1 shows the meta-model of finite-state machines. Compare it with Tbl. 3.1 when reading. In the figure, we have classes representing finite-state machines, states, and transitions. The Model class allows us to have a single object as a handle to several state machines in a model.

A containment association states (black diamond) captures the part-of relation between a state machine and a state. Even though transitions can be thought of as relations, we model them as first-class objects. This is because we need to store properties (the input and output labels)—associations

Ecore, MOF, and Meta-Modeling

Meta-Object Facility (MOF) is a simple class-modeling language standardized by the *Object Management Group* (OMG). MOF was created as the meta-modeling language to be used in writing the UML standard. It is used by OMG to define the meta-models of the UML sub-languages. MOF is a minimalistic class-modeling language that is relatively easy to learn and implement. MOF includes packages, classes, attributes, simple types, containment, operations, multiple inheritance, interfaces, and binary unidirectional associations (references). It excludes advanced constructs of UML Class Diagrams, for example *n*-ary associations and association classes. You can inspect the freely accessible MOF specification at https://www.omg.org/spec/MOF to get an idea what a formal modeling-language standard looks like.

Ecore (https://www.eclipse.org/modeling/emf/) is the Java implementation of the MOF specification by the Eclipse Modeling project. Ecore is used for meta-modeling in the Eclipse Modeling Framework (EMF). Meta-models in Ecore are compatible with EMF's rich tool ecosystem, which can be used to implement your DSL. Like MOF, Ecore is used for meta-modeling, so with exactly the same purpose the MOF designers had in mind: to build language models. In fact, when we specify DSLs, such as state-machine languages and configuration languages in Ecore, we follow the same method that UML designers used to specify the abstract syntax for all UML diagrams, including the state-machine diagrams.

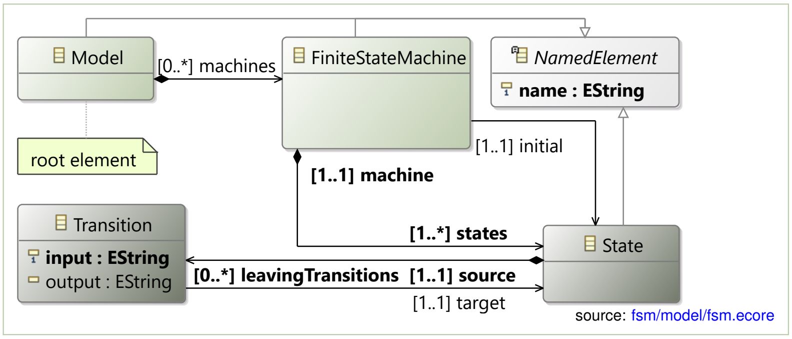


Figure 3.1: A meta-model for the language of finite-state machines using class diagrams as the meta-modeling language. Compare with Fig. 5.10 on p. 164

cannot carry attributes in Ecore. The source and target references represent the connection relations between a transition and its incident states. In order to ensure that the entire instance is a tree (a single *partonomy*, see below) we made the source relation a containment (black diamond again).

The transition objects are contained in the source state—this makes execution of machines easier; containment references are navigable. This is an example when implementation considerations pollute the meta-model—a pragmatic compromise that allows the same model to be used for domain analysis and for implementation. Such compromises are often made for simple languages.

We decided to make initial a relation between a state machine and one of its states. Alternatively, we could have modeled the initial state as a Boolean attribute of one of the states. What we did requires a constraint that the initial state of a machine is actually one of its own states (so the initial relation is a subset of the states relation). The alternative modeling requires a constraint that exactly one state in each state machine has the initial attribute set to true,

```
p {
  background-color: black;
  color: blue;
}

div { background-color: red; }
```

Figure 3.2: *An example CSS file for Exercise 3.2*

and all others are set to false. We discuss how to add constraints to meta-models in Chapter 5.

We use a convention, instead of an explicit meta-model element, that any state without an outgoing transition is an end state. Not declaring this property in the meta-model makes it more concise, but users of our meta-model (developers implementing transformations) need to be aware of this convention, which is not necessarily obvious, especially since graphical state-machine notations often have a dedicated symbol for end states.

Since state machines and states both have the attribute name, we extract it to an abstract class *NamedElement*. This is a common pattern in object-oriented meta-models. It allows a single visualization code to be used to label objects that are named. For example, the EMF framework itself interprets this attribute in a special way, displaying names as object identifiers in editors.

As shown in the above example, domain analysis is transferred to meta-models naturally, like in most other examples of object-oriented design. Most formal meta-models, and especially those based on class diagrams like our example, capture the answers to the third and fourth question of our simple domain-analysis scheme from Sect. 3.2 (concepts and relations), but they are heavily influenced by the collected examples and use cases (we come back to this in Sect. 3.8).

Exercise 3.2. Figure 3.2 presents an example domain-specific model in the CSS (*Cascading Style Sheets*) language. Assume that a CSS model consists only of top-level style specifications for elements of type `p` (paragraph) and `div` (document sub-tree). These can be repeated arbitrarily many times, and mixed in any way. Each specification can contain an arbitrary number of attributes in any order, but there are only two kinds of attributes: `background-color` and `color`. Each of these properties must have a color assigned selected from the list: `black`, `white`, and `red`. If you know any other aspects of CSS ignore them for now, to simplify the task. Design an Ecore meta-model (or a set of suitable Scala types) for representing this subset of CSS. Name the root class of the CSS meta-model.

Exercise 3.3. Refactor the meta-model presented in Fig. 3.1 so that 'initial' is a Boolean attribute of a State, instead being a (non-Boolean) property of the FiniteStateMachine.

Instances of meta-models. Meta-models define all possible models and therefore all possible instances of a language in abstract syntax, disregarding the particular textual or graphical notation. These instances become in-memory objects in tools such as the language editor, serializer, and deserializer. They are also processed by model transformations, which we discuss in Chapter 7.

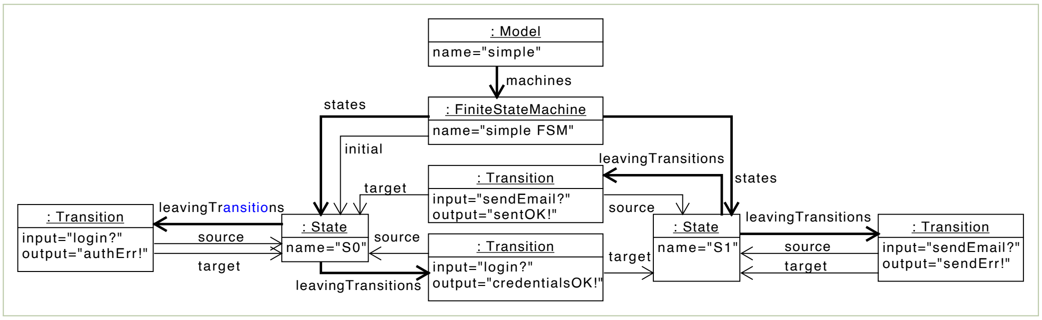


Figure 3.3: An instance (in abstract syntax) of the finite-state-machines language defined by the meta-model in Fig. 3.1

Instances quickly become large, when their visualization is difficult, impractical, or impossible. Furthermore, in the context of MDSE and DSLs, we typically do not need to show instances in a generic notation—we use concrete syntax of the DSL instead. However, for learning and understanding, it does make sense to take a look at an instance to develop an intuition for how instances and meta-models relate. Fig. 3.3 shows a simple instance in abstract syntax of our language for finite-state machines. It corresponds to the instance shown in concrete syntax in Tbl. 3.1. We use a UML instance specification diagram to visualize this example.[1] It is instructive to compare the concrete state machine in Tbl. 3.1 with this figure, and with the meta-model of Fig. 3.1. The concrete syntax of the example has two states, and the object diagram has two State objects (S0 and S1), while the meta-model had a single State class. Similarly we have four transitions (arrows) in the concrete syntax, each represented by an object, an instance, of the Transition class of the meta-model.

Exercise 3.4. Draw the instance representing the example of Fig. 3.2 as an instance (object diagram) of the meta-model designed in Exercise 3.2.

3.4 Guidelines for Meta-Modeling with Class Diagrams

In general, all design patterns and analysis methods known from class modeling apply to meta-modeling. However, the meta-modeling use case has its few specific requirements that lead to some specific design recommendations and patterns. Let us discuss these now.

Create a single partonomy. A partonomy is the decomposition of a class diagram along the part-of relationships. *Meta-models should have a single partonomy*, so in each instance every object should be contained (perhaps indirectly) in the containment hierarchy of a single root element. Basically, there should be a single connected syntax tree. ***Guideline 3.1***

[1]The appendix “Class Modeling” on our book website (http://dsl.design) talks about instance specifications. An “object diagram” was a diagram type available in older versions of UML, before UML 2.0. Objects are now called instance specifications and integrated into class diagrams. The notation remained the same, however.

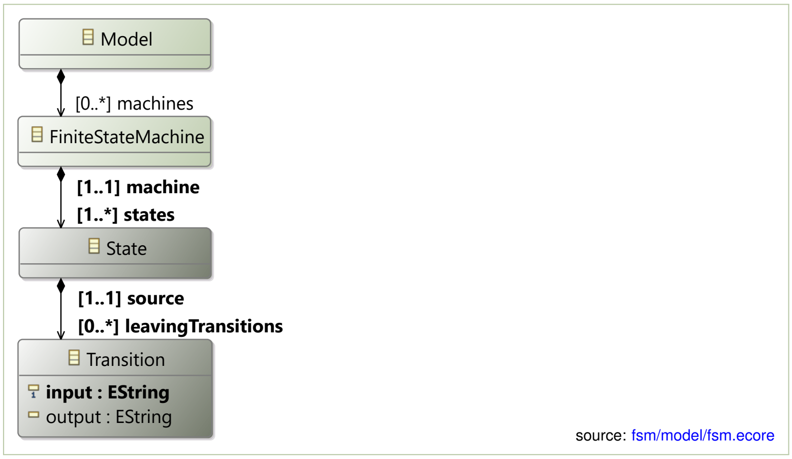


Figure 3.4: *The partonomy of the meta-model of Fig. 3.1*

The partonomy of the diagram in Fig. 3.1 is shown in Fig. 3.4. A partonomy view of a diagram shows the decomposition of structures: a sub-diagram showing the classes and their containment relationships. In our example, the decomposition is a very simple nesting (transitions are nested in states, states are nested in finite-state machines, and machines are nested in models). In general, a partonomy takes the form of a forest. In meta-modeling, we introduce a class as the top-level node, usually representing the model or the document, that owns all the forest's trees. This way, we arrive at a single tree structure. This structure is then easily manipulated in programs, where it can be passed around and accessed using a single root object. It is important that all classes are transitively contained by the top-level class. Otherwise, the class could not be instantiated; more precisely, it could, but would not be contained by another object and therefore immediately deleted by the garbage collector of the underlying programming language (e.g., Java).

Exercise 3.5. Draw the partonomy view of the meta-model created in Exercise 3.2.

Guideline 3.2 *Avoid interfaces and methods. It is a bad smell if you see interfaces or methods in your meta-model.* Inexperienced modelers often confuse abstract classes and interfaces, presumably due to the relative interchangeability of these in programming. Abstract classes represent abstract concepts and properties in the meta-models and are related to concrete concepts using a *kind-of* relation (generalization); for instance, the abstract class NamedElement in our example. In contrast, interfaces, as opposed to abstract classes, are meant to represent the APIs of objects with which you or others are interacting.

Methods (operations) rarely appear in meta-model classes. The MDSE tools that process instances and meta-models only take the structure, qualities, and relations into account, not the methods. At runtime, the in-memory objects instantiating the meta-model classes tend to be passive. Any operations on them are usually implemented outside the generated classes, in the interpretation and transformation modules. For these reasons, you should normally not place methods and interfaces into class diagrams that are meta-models—not least to avoid becoming confused about the role of meta-models in the MDSE process.

Yet, a few exceptions to this rule exist. Methods can become handy if you want to create *derived attributes* (properties that are computed based on other properties and relations). In this case, you can put a respective method into a class in the meta-model. Beyond derived properties, sometimes it is too much overhead to separate simple behavior. Then, it might be useful to implement convenience operations directly in the generated classes. For instance, the state-transition logic for our finite-state-machine example could be implemented as additional methods in the generated class State.

> **Exercise 3.6.** Write a Java (or Scala) function `isInitial` that should be a member of the class State (Fig. 3.1). The function should return true if and only if the `this` object represents an initial state. Note that `isInitial` is a derived attribute, and its representation in the diagram is actually shown later, in Fig. 13.2 on p. 462.

Adding convenience methods in the meta-model and in the generated code can be entirely avoided when you are using a sufficiently expressive programming language. For example, extension methods (C#, or Xtend) can be used to provide these methods outside the generated code. In Scala, implicits are used to add extension methods, following the *pimp my library* pattern—especially for Java libraries, such as the code generated by EMF. In AspectJ or Kermeta, aspect weaving can be used to achieve a similar effect. In all these cases, we get an architectural advantage of keeping all hand-written code in separate compilation units from generated code. This simplifies incremental builds, error-reporting, and rerunning test cases, and reduces the risk of manually modifying generated code, which introduces a slippery slope of abandoning all benefits of MDSE in the long term.

Finally, the well-established Model-View-Controller pattern [14] calls for separating operations (and visualization) on the model from the model itself. This pattern is commonly followed in MDSE. When no operations are put into the model, the chances of violating this pattern are much lower.

Verify the taxonomy. The taxonomy of a meta-model is the way concepts are classified, and how classes are organized in a hierarchy. In object-oriented meta-modeling the taxonomy is naturally given by the generalization (inheritance) hierarchy of classes. A taxonomy view can be produced from your diagram by removing associations and only retaining the generalization relation and classes. Unlike for partonomies, quite often there is no single taxonomy in a meta-model, but several disconnected ones. ***Guideline 3.3***

> **Exercise 3.7.** Draw the taxonomy view of the diagram in Fig. 3.1. Recall that the partonomy view of this diagram is shown in Fig. 3.4.

In the case of finite-state machines, the taxonomy view is somewhat simplistic. The only generalizations in the diagram that might not be recognizable for domain experts involve the abstract class NamedElement. In general, however, the taxonomy view will show a useful decomposition of the concept space that is dual to the partonomy decomposition. *It is useful to verify the taxonomy of the meta-model with domain experts.*

For instance, if we model embedded-system components, we may see generalizations between less and more advanced versions of a component. We may also see abstract classes (and generalizations involving them) representing component categories. Such a taxonomy should appear familiar to domain experts and may be verified by them. Incidentally, the ability to express concept taxonomies is one of the key advantages of class diagrams over relational schemas for domain modeling. Entity-Relationship (E/R) diagrams feature only relations (associations) between concepts, without any way to express generalization first class.

Guideline 3.4 *Reify relations when necessary.* If relations between concepts have properties, you can reify them as classes. Even if you use a simple modeling language such as Ecore, which does not use association classes, you can represent relations as classes, not associations. We have seen this pattern at work in the state-machine example, where the transition could alternatively be modeled as a successor relation between states. Turning an association into a class and two relations for each of the original endpoints allows us to place the attributes on the class, which was not possible for Ecore associations.

Exercise 3.8. Redesign the state-machine meta-model to use an association (reference) instead of a class for transitions. While doing this, simply ignore the input and output labels—drop them from the meta-model.

Guideline 3.5 *Avoid redundancies.* It is a bad smell if multiple classes have the same attribute. If multiple concepts share an attribute (name, size, speed, etc.), then it is very likely that in your implementation of the framework you would like to perform common operations on them (e.g., printing, measuring, moving). This will be easier to do in a reusable way if you extract the common properties to abstract classes, like we did with the name attribute and the abstract class *NamedElement*.

Guideline 3.6 *Use singular for class names.* It is usually a bad smell if a class name is in plural. Recall that you describe the main concepts in your domain and their relationships, including how many instances of which concept are in a relationship with how many other concepts. To precisely express this, each class should represent one concept, and a concept is typically expressed in the singular (e.g., Person or Customer), only very rarely in the plural (e.g., Statistics, CustomerServices).

3.5 Meta-Modeling with Algebraic Data Types

From the programming language point of view, meta-models are just definitions of classes and properties. They are types, basically. We have shown how to use Ecore to express meta-models, but most modern programming

Opposing Forces in Meta-Modeling

A DSL meta-model is a technical artifact that responds to opposing forces. As a pivotal artifact in a project, it needs both to capture key aspects of the input domain and to provide types for instantiation and manipulation used in the implementation. For example, its instances should be easy to construct using a parser (Chapter 4) and easy to navigate to required elements in interpreters and code generators. This means that compromises are often made.

Fowler and Parsons [8] often find it helpful to consider how the instances are supposed to be used by the software framework, when designing the meta-model. So they take both the domain (end-user) perspective and the implementer perspective into account. Often the most elegant model from the domain perspective is not the most convenient from the implementation perspective. If the gap between the problem-space requirements and the solution-space needs are too large, it is not unusual to work with two meta-models: one that is close to the domain (a so-called *Platform-Independent Model*) and one that is closer to the solution (a *Platform-Specific Model*). In this case we can use a model-to-model transformation (Chapter 7) to translate between the problem-space model and the solution-space model.

languages have sufficient facilities to express similar information directly, without using Ecore. Thus you have a choice between using a dedicated modeling framework or modeling directly with types. In this section, we present the functional-programming style of abstract-syntax definitions—a popular meta-modeling alternative among language designers.

Example 8. Recall the meta-model of finite-state machines of Fig. 3.1. Fig. 3.5 shows how the corresponding Scala case classes, an algebraic data type, look. Read the figure and compare it against the class diagram, before proceeding. Below, we comment on the six types defined therein: `NamedElement`, `ModelElement`, `Model`, `StateName`, `Transition`, and `FiniteStateMachine`.

`NamedElement` is a Scala trait (similar to a Java interface, but it can also carry attributes). Like in the Ecore meta-model, we require that all named elements have a string attribute `name`. Since traits support multiple inheritance, this modeling corresponds directly to the use of the abstract class NamedElement in Fig. 3.1, where NamedElement was also used in multiple inheritance of Ecore.

We could use a getter-and-setter pattern to define the name attribute—EMF does this when generating code from our Ecore meta-model. However, in the pure functional-programming setting here, public access to values is much less of an issue than in classical object-oriented programming. Since attributes cannot be modified, invariants are not easy to break. For this reason, publicly accessible read-only fields, without a getter and setter, are common in functional programming. Violation of access is less of an issue there, as pure code, without side effects, cannot break data invariants, even if accessing values directly. In turn, we gain conciseness and simplicity of the definition—less coding and less maintenance. Still, it might sometimes be valuable to hide properties behind access methods in functional programs—to prevent external code from developing dependencies on internal representations. This is relatively rarely used in language engineering, as it is in any case hard to evolve language

```scala
trait NamedElement:
  val name: String

sealed abstract trait ModelElement

case class Model (
  name: String,
  machines: List[FiniteStateMachine]) extends ModelElement, NamedElement

type StateName = String

case class Transition (
  target: StateName,
  input: String,
  output: String = "") extends ModelElement

case class FiniteStateMachine (
  name: String,
  states: List[StateName],
  transitions: Map[StateName, List[Transition]],
  initial: String) extends ModelElement, NamedElement
```
source: fsm.scala/src/main/scala/dsldesign/fsm/scala/adt.scala

Figure 3.5: *Another modeling of the finite-state machine language, using Scala (cf. Fig. 3.1)*

syntax implementation without changing the abstract API, so the external code depending on the API would break anyway when the language evolves.

Returning to the figure, `ModelElement` is an abstract type that we use to designate all program classes that are part of the meta-model. It corresponds roughly to the EObject type defined by Ecore, which is a super-type for all instance objects at runtime (it is defined in the Ecore library and used in the code generated from the meta-model). With use of the `ModelElement` type we can later write generic code that processes any kinds of instance objects, while still being type safe. As you notice, there is slightly more work to do in bare-bones Scala than in Ecore—EObject was defined once and for all in Ecore, here we do the work in the meta-model.

The `ModelElement` trait is sealed in this example, which limits the possible implementations to the three types defined below in the same file (`Model`, `FiniteStateMachine`, and `Transition`). We seal this trait to emphasize that the listing in Fig. 3.5 contains the complete meta-model definition—no further extensions will be done in other files. It also allows the type checker to warn the programmer, whenever a type-based pattern matching expression neglects one of the three cases. The three classes correspond directly to the concepts of model, state machine, and transition in the Ecore meta-model.

An attentive reader has already noticed that we lack a class definition for State in this example. For simplicity, we represent states directly by their names (character strings). We introduce a type alias `StateName` purely for readability. While modeling states as a class was entirely possible in Scala, we decided to choose another route to illustrate an alternative pattern, creating a meta-model where the transition relation (the `transitions` property in `FiniteStateMachine`) is a single first-class data structure, not distributed by different containing states. Such representation of transitions as a single

```
me match
  case Model (name, machines) =>
      ... // code executed if me is an instance of Model
  case FiniteStateMachine (name, states,t ransitions,initial) =>
      ... // code executed if me is a FiniteStateMachine
  case Transition (target, input, output) =>
      ... // code executed for transitions
```

Figure 3.6: *A Scala pattern-matching expression for the types of Fig. 3.5*

relation is a common functional modeling style for automata-like languages. Modeling as a relation (a set of arrows) is also a typical way to represent cyclic structures (which finite automata are)—otherwise there is no way to construct cyclic instances in a purely functional manner in strict languages like Scala. A key decision was to lift the transition representation from the state to the machine level, as a structure over states, not part of each state. Using a map is secondary; we could have used an association list, or other structures.

Algebraic data types. In Scala, we implement *algebraic data types* (ADTs) using sealed traits and case classes. An ADT comprises a number of type cases (a union, a sum of cases), where each case combines a tuple of several attributes (selected from a Cartesian product of types). The name *algebraic* stems from this combination of two operators to generate the type extension: the union and the product. ADTs in other functional languages are also known as data types, union types, discriminated unions, tagged types, etc.

Why do we choose to model abstract-syntax trees (and so, meta-models) with ADTs? First, the combination of products and sums allows us to represent trees of diverse shapes. The branching degree of a tree node is defined by the arity of the cases class representing this type of node. This makes ADTs extremely practical for specifying abstract-syntax trees, which *are* trees of irregular arity. Second, ADTs combine well with pattern-matching expressions (switches over types), which allow us to concisely write language-processing algorithms, especially interpreters and transformations. Fig. 3.6 sketches a skeleton of a type-safe, statically checked code that generically processes any kinds of model elements in our example.

ADTs in functional languages without inheritance (e.g., Haskell, Standard ML) can typically capture only the partonomy view of a meta-model. The taxonomy needs to be worked around in code, using the available reuse mechanisms (e.g., type classes, functors). In languages that combine algebraic data types and classes (e.g., Scala, F#), most of the meta-modeling goes in the same way as in Ecore: we map the taxonomy to the inheritance hierarchy, and the partonomy to the nesting of type properties. Unfortunately other relations (that are not guaranteed to be acyclic) are impossible to represent directly in an immutable ADT. The ADT constructors can only build trees—to close a cycle we either need to use a side effect (an assignment) to redirect a reference, or we have to resort to indirect modeling. A classical solution is to use name-based references. In the

```
val transitions = Map (
  "S0" -> List (
    Transition (input="login?", output="credentialsOK!", target="S1"),
    Transition (input="login?", output="authErr!",       target="S0")),
  "S1" -> List (
    Transition (input="sendEmail?", output="sentOK!",  target="S0"),
    Transition (input="sendEmail?", output="sendErr!", target="S1"))
)
val machine = FiniteStateMachine (
  name="simple FSM",
  states=List ("S0", "S1"),
  transitions=transitions,
  initial="S0"
)
val model = Model ("simple", List (machine))
```

Figure 3.7: The instances of Fig. 3.3 recoded as an instance value for types in Fig. 3.5

example, elements' names (strings of characters) are used as identifiers, and we replaced references to objects with references to their names.[2]

Since we are now using name-based references, we also need a dictionary that maps names to objects. We have two such dictionaries in our model: one is the list of state names and the other is the map from state names to the outgoing transition lists. Typically the dictionaries need to be computed separately, after parsing is completed. It is clearly an advantage of Ecore and of language workbenches such as Xtext that they perform this work for you, relinking all the references in the object graph after parsing and type checking.

> **Exercise 3.9.** Design a Scala algebraic data type representing the same information as the meta-model of CSS created in Exercise 3.2.

Ecore instances are serialized to XMI files. We do not have such a generic facility for regular values of programming languages (although some languages offer marshaling libraries to JSON, YAML, XML, or custom binary formats). If you need a text representation that requires no additional infrastructure, the easiest way to create and save instances of an ADT meta-model is writing constructor expressions directly in Scala. Figure 3.7 shows the instance of the finite-state-machine language originally presented in Fig. 3.3 written as an instance of Scala types of Fig. 3.5. This is probably the easiest way to store and reuse instances in testing. If you need to use other modeling tools that rely on the XMI Format, you will have to implement a suitable transformation first.

To simplify the construction of instances as constructor expressions, we often implement default parameter values (see `Transition.output` in Fig. 3.5, not exploited in Fig. 3.7), alternative constructors, and factory methods. Language elements tend to have a lot of optional properties. Providing all of them explicitly at instantiation quickly becomes burdensome.

[2]Hint: Use strings of characters as element identifiers for languages that have a single name space and small models (human created). In such cases, strings are sufficiently efficient and tend to be the simplest way to implement references, as most often you only need to make cyclic references to elements that already have names. The added bonus is that instances are easy to read and reasonable to write for humans, for instance during testing and debugging.

Exercise 3.10. Write down the scala value representing the abstract syntax of the example CSS instance in Fig. 3.2. Use types and constructors defined when solving Exercise 3.9.

ADTs vs Ecore. So what should I choose: a modeling DSL from a language workbench like Ecore or just standard ADTs from my programming language? One advantage of Ecore, and other language development frameworks, is that many tools will integrate with their representations, even from other programming languages than Java. If you need any Ecore (or XMI) dependent technology, we recommend using Ecore for modeling. At the same time, it should be noted that modeling and processing abstract syntax of languages was one of the key motivations for creating modern functional programming languages. In fact, the ML language, a predecessor of Standard ML, OCaml, F#, and Scala, was originally created to build a proof assistant in which logical statements had to be easily represented and transformed using inference rules (so this was a language-engineering project!). Ever since, functional programming languages have been popular with language researchers and nowadays also with the industry developing languages. This means that much competing infrastructure exists in the ecosystem of functional programming languages.

Ecore meta-models tend to have a better representation of constraints than ADTs. In most languages, ADTs do not provide modeling facilities for capturing cardinality constraints, or *bidirectional* associations (so associations that can be navigated from both ends, unlike regular references in programming languages that are unidirectional). This is why you will be reimplementing some of these facilities manually, often by writing additional static-checking code. On the other hand, functional programming languages offer a fairly concise programming style, well suited for language processing, that comes in very useful in later language development stages. The good news is that this style can also be used with Ecore (with some friction due to the imperative nature of Ecore's APIs). Programming languages like Scala, Xtend, and recently also Java, allow functional programming to be used with Ecore generated types.

Using a programming language in domain analysis tends to quickly bring us into fairly low-level technical discussions (as seen to an extent in the `fsm` example). While this is not immediately a problem for experienced language engineers, if you are new to language design you might find this design stage unduly daunting when using ADTs. Regardless of which technology you use, you can follow the domain analysis process outlined earlier in this chapter. This book allows you to explore, compare, and reflect about both worlds, hopefully leading you to a much more informed choice. Even if you only use one of these technologies in any given project, knowing how languages are designed and implemented across technological spaces should make you a better language engineer.

3.6 Language-Independent Meta-Modeling Guidelines

To further demystify the process of meta-modeling, we present modeling advice collected from teaching experience and published research works. The guidelines below apply regardless of whether you use object-oriented syntax modeling or algebraic data types.

Guideline 3.7 *Let the meta-model describe the problem, not the software tool solving it.* The similarity of meta-modeling with the design of object-oriented APIs tends to confuse inexperienced language designers. It is key to understand that meta-modeling is not programming of your tool infrastructure, and the concepts in the meta-model are not the components of your tools! When we use class modeling for creating meta-models, a class primarily represents a domain concept, and not an implementation concept. The model you are building is the model of the language, not an architectural diagram of your tool, as often seen in introductory object-oriented modeling courses. So, while we have states and transitions in our example, we do not include the parser, the interpreter, a code generator, or a type checker in the meta-model. These are not part of the language, but of the surrounding infrastructure.

Guideline 3.8 *Avoid scope creep.* Design what is absolutely necessary and avoid natural tendencies to over-design [24]. Regardless of what meta-modeling language you use, the meta-model should have as few concepts as necessary, and no more. You can get there by questioning everything in the language design; specifically, question why each construct is needed, and why already now, in the current release of the language. Wile [24] suggest to focus on about 80% of the needs, and to provide a way to escape outside the DSL, or to extend it programmatically in the underlying system, for the rest of the complex cases. Once the language is created and the infrastructure is implemented and used, it is expensive to revert decisions. A smaller language is not only cheaper to maintain and evolve, but also faster to learn. Releasing in small increments allows tests with users to be run earlier and thus lowers risks.

Guideline 3.9 *Use abstraction wisely.* Even though abstraction is nice, always think what level of abstraction and detail is sufficient and necessary. Consider for instance our `fsm` language. When modeling execution time in a state machine, you can decide between having the time in seconds versus just using labels fast/slow. The latter might be sufficient for some applications, such as a coffee machine that has a fast and a slow brewing mode. You could also simplify the language, i.e., abstract it, by having fewer labels on the transitions. Drop the inputs when you notice that the language is just used for specifying behavior in terms of actions and does not need to react to input.

Guideline 3.10 *Strive for simplicity.* A language must be simple [12, 11]. Implementing a complex language will use a lot of resources. Keep the number of concepts as small as possible and avoid redundancy (i.e., the language's ability to express the same things in many ways). Also accept that your language will be incomplete. It is dangerous to create a language that covers all possible

Grammar-First or Model-First?

Some authors suggest that the language design should start with the concrete syntax and should use meta-models (or other abstract-syntax definitions) as secondary implementation artifacts [20, 13]. In this book, we advocate designing a domain model first, before developing concrete syntaxes. We believe that the meta-model is a central, pivotal artifact in several ways. First, constructing the meta-model is an instrument for performing domain analysis and problem understanding. Thus it is key for system design. Second, designing the meta-model helps you to avoid the trap of jumping to solutions too quickly, stay longer on the problem, and avoid being driven early in design by ad hoc concrete syntax ideas. Third, different elements of your tool chain will communicate using the instances of the meta-model. These different parts need to be able to query and manipulate the instances efficiently and effectively. Automatically generated meta-models are usually far from natural and far from elegant. If you use them, you need to program with convoluted types and APIs, and as a result your back-end tools become complex.

general cases. It is more important to create a language that covers cases appearing in practice and then plan for language evolution.

Prepare the language to grow. Making the language simple is only safe if you take some protective measures against trivialization. First, be ready to grow the language iteratively in the future [4]. Few languages never need to be evolved. DSLs are like libraries and need continuous growth. Second, consider making the language open—equip it with an escape mechanism, so that users who outgrow the language can circumvent its limitations [4]. This can be done at various phases of the language design. In domain analysis this may require considering an escape construct to call lower-level code. Alternatives include implementing the language as an internal DSL (Chapter 10), or providing an API to hook into your interpreter or code generator. **Guideline 3.11**

Avoid designing programming constructs. It is usually a bad sign if your language becomes dominated by typical programming constructs such as loops, branching, functions, and classes. *You are almost never designing a programming language* [24]. This is often a sign that your abstraction is not close enough to the domain. It is better to stick to the problem domain as closely as possible [12, 21]. However, if your language is meant to describe large, complex systems, *consider adding modularity constructs* to it. Large models need to be broken into smaller pieces [11]. **Guideline 3.12**

It is better to *use the problem domain as inspiration*, rather than the solution space [12]. This applies *even* if an implementation exists, such as in re-engineering scenarios where models and code generators are introduced into an existing system. Of course, one should be realistic and still design a language that can be realized on top of an existing framework. Solution-space constraints should not dominate the design, however.

3.7 Case Study: Mind Maps

We shall now pursue a larger example to illustrate the domain-modeling and analysis process. Our hypothetical goal is to build a mind-mapping

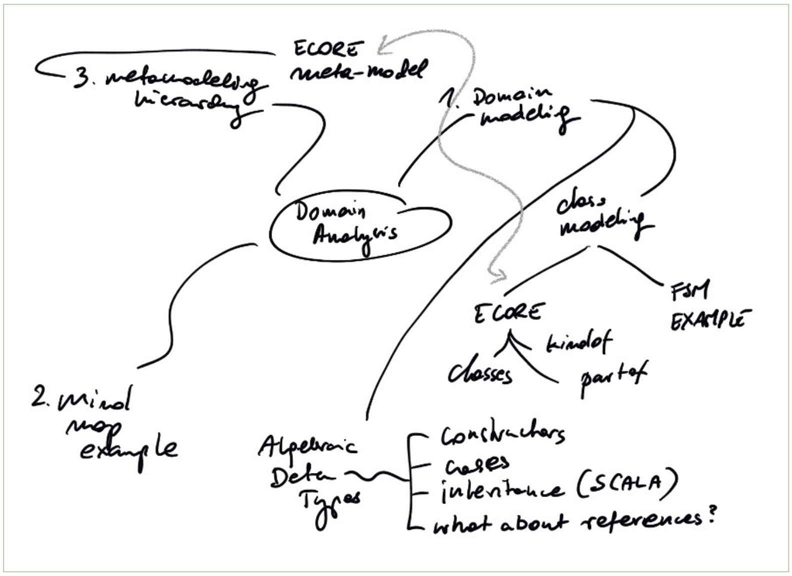


Figure 3.8: An example mind map, hand-drawn on paper

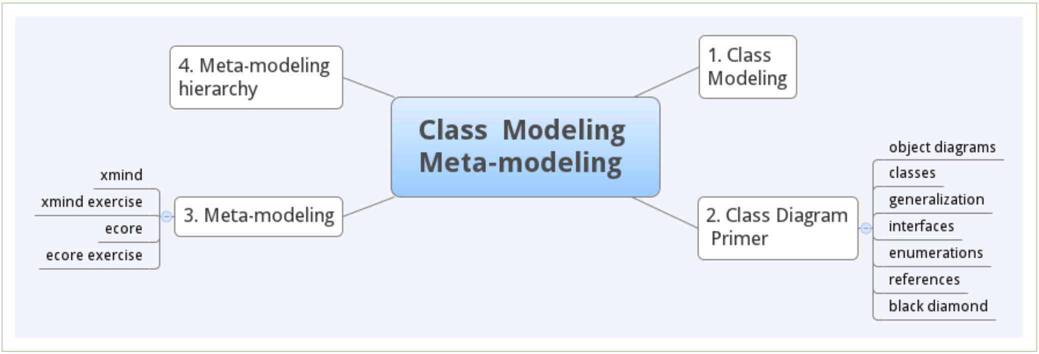


Figure 3.9: A mind-map model shown in concrete syntax (created by the program XMind)

tool, not unlike XMind[3] or FreeMind.[4] A mind map is a diagram that organizes information visually. Each mind-map diagram has a central concept, usually represented by a label centered on a page, from which a hierarchy of concepts and ideas extends concentrically. Fig. 3.8 presents an example mind-map diagram that could have been created while taking notes during a lecture on meta-modeling. Our goal is to create a modeling language that allows us to draw mind maps on a computer.[5]

Figure 3.9 shows a piece of concrete syntax of an existing mind-mapping tool. We discuss the domain analysis in Tbl. 3.2. Figure 3.10 shows a potential meta-model of this mind-mapping language. At this point, there

[3] http://www.xmind.net, retrieved 2022/08

[4] http://freemind.sourceforge.net/wiki/index.php/Main_Page, retrieved 2022/08

[5] The example in this section is loosely inspired by a blog post of François Pfister, available at: http://gmf-modeling.blogspot.com, retrieved 2022/08

Purpose	To be able to take simple lightweight notes in a mind-map format. To be able to read these notes, when studying for exams.
Users	Students taking notes during lectures and during exam preparation.
Concepts	The center of the diagram contains the main topic, which is then also the root of the note's hierarchy. Subtopics are organized centrally around the main topic.
Relations	*Properties:* The topics can be (optionally) numbered to indicate the order of reading. Some topics can be emphasized (for instance printed with a bold font). Finally, topics are indexed by colors, so that tools can mimic the idea of using several pens, when displaying the nodes. *Relations:* The key relation is that between topic and subtopic: decomposition of the topic into subtopics. The nesting using the decomposition relation can be arbitrarily deep. Besides this decomposition it is also possible to draw lines between topics that are related even if they are not neighbors in the topic decomposition hierarchy.
Examples	Fig. 3.8 provides an example. We note that in this example topic decompositions are black, and the cross-hierarchy relation is drawn in a light gray color. In this example only the first layer of topics around the center is numbered (the other topics are not numbered). The hierarchy is five topics deep, and only black color is used. The use of syntax is informal, and at places inconsistent as is common for notations used for sketching or brainstorming on paper, before they have been formalized.

Table 3.2: Knowledge collected in a hypothetical domain analysis process for the mind-map example

should be nothing surprising in this meta-model. Still, let us discuss a specific design decision. The class Color was designed to be contained by the class Model. Each topic has optionally a reference to a specific color. We could have let Color be contained by Topic, but since multiple topics will likely have the same color, we would need to instantiate the same color multiple times. We avoid this redundancy by this containment hierarchy, so only one Color instance shall be created per unique color code (attribute Color.rgbcode). Note that you could still create multiple Color instances with the same color code; nothing prevents you from doing that. This issue will be addressed using constraints later in Chapter 5.

Another issue with the class Color is that, when instantiating colors, you need to create the color code, a string containing three hexadecimal numbers representing the values for the red, green, and blue parts. It might be better to predefine potential instances representing known colors directly in the model. Unfortunately, that is not possible in Ecore. UML has a suitable construct, called *instance specification*, which we will return to in Sect. 3.9.

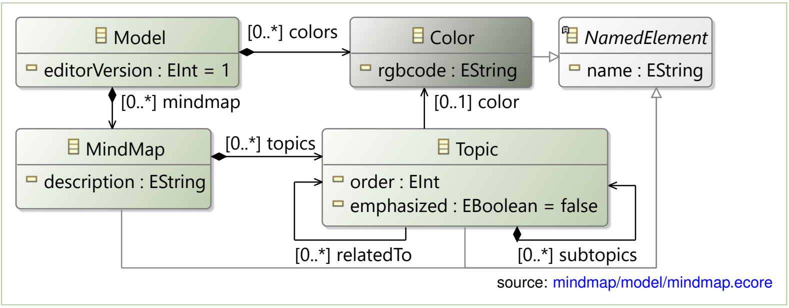


Figure 3.10: The meta-model of a simple mind-mapping language

3.8 Quality Assurance and Testing for Meta-Models

In MDSE, meta-models become pivotal elements, used by all other parts of your tool chain. It is thus key that they are correct. There are two main quality assurance (QA) objectives for meta-models: first, confirm that the meta-model meets the requirements of the project (can we describe everything we need?); second, ensure that the model has good quality and contains no design errors. Let us discuss the strategies to achieve these goals.

Meeting the requirements. The key and too frequently neglected QA practice is *checking whether the meta-model adheres to its requirements.* We recommend a systematic and regular review of the requirements against the meta-model. For example, if you followed the method of Sect. 3.2, you can revisit the collected material in the QA phase: (i) Check whether the purpose of the language has not moved from the prescribed goal. (ii) Check whether the concepts in the meta-model remain relevant for the stakeholders. (iii) Check whether the relevant concepts and relations from the requirements are reflected well in the meta-model.

Definition 3.3. *A meta-model is* complete *if its instances can represent all the domain problems as defined in the system requirements.*

The check for completeness should be organized not by model elements, but *by the requirements*. A reasonable stop criterion for the activity is thus achieving *high coverage of requirements* or simply establishing that all the requirements are met. If you work through all the model elements, you will not be able to see whether your model misses some aspects.

One way to make the completeness check concrete and focused, while producing useful artifacts, is to manually create the instances of the domain model that witness meeting the requirements. Bentley [4] recommends: "Before implementing the language, test your design by describing a wide variety of objects in the proposed language." You can use the concrete cases collected in the domain analysis as an inspiration for some of this work. The created instances should be saved in the language development repository as test cases for development of other language aspects. They will be instrumental in setting up automated tests of the implementation of static and dynamic semantics, and as oracles for testing the front-end. They will also

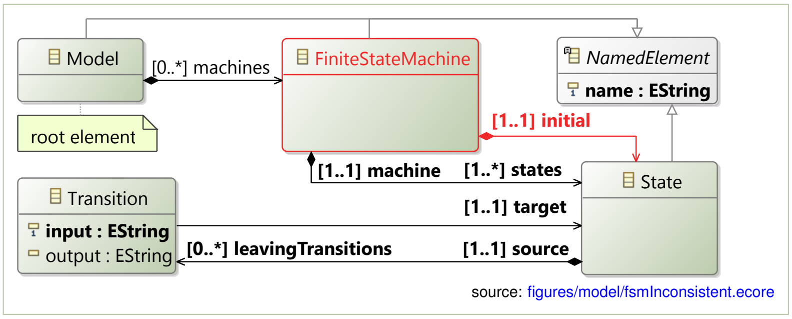


Figure 3.11: A version of the meta-model for the language of finite-state machines that is consistent, but not element-consistent

allow the front-end and the back-end to be developed and tested separately, which is important for parallelizing work: the concrete syntax developers can use them as oracles for results of parsing, the static semantics developers and code generator developers can use them as initial test subjects.

Involving domain experts in the process is an advantage if possible, but they might prefer to communicate using concrete syntax (see Chapter 4), so the test with language users may be better done slightly later, or using concrete syntax mock-ups.

Internal quality of the meta-model. Independently of assessing the extent to which the meta-model meets the requirements, it is worthwhile to check the internal quality of the meta-model. We focus on two main criteria: consistency and parsimony.

Definition 3.4. *A meta-model is* consistent *if it can be instantiated meeting all constraints of the meta-modeling language semantics. A meta-model is* element-consistent *if for each element of the meta-model there exists an instance in which this element is instantiated.*

Figure 3.11 presents a minor (erroneous) variation of the meta-model for finite-state machines originally presented in Fig. 3.1. Only one property is changed: the initial reference is turned into a containment (highlighted in red). This meta-model is consistent, but not element-consistent. It can be instantiated by creating an instance of the Model class without any machines. Instantiating the FiniteStateMachine class is not possible. Observe, that a FiniteStateMachine object should contain a State object, but a State object must be *contained* both in the states collection and in the initial property, which is not possible simultaneously. The meta-model can be fixed by *relaxing* the containment constraint or by relaxing the cardinality (to make both containments optional).

Inconsistency is always a manifestation of an internal quality problem in a meta-model. An inconsistent meta-model is useless. It defines an abstract syntax for an empty language. An element-inconsistent meta-model can only be partially instantiated. It is not useless as a whole, but it has parts that are useless. Inconsistent elements in a meta-model are like dead code in programs—most often a manifestation of problems as well.

It is fairly rare that meta-models created by experienced modelers are inconsistent (or element-inconsistent), but inconsistency errors often show up in models created by beginners. Thus, you should use consistency testing also as a way to learn meta-modeling. One group of consistency errors emerges from the interplay of containment and cardinality constraints (like in our example). It is also possible to create inconsistencies by building collections of enumeration values with cardinality constraints and a uniqueness constraint. For example, there may be not enough values of an enumeration type to populate a collection with unique elements, and to satisfy a lower bound on the size. Finally, inconsistency errors can also arise in an incorrect construction of your partonomy (a disconnected or circular partonomy)—recall that all meta-classes must be reachable from the root model class through partonomy links, so they can be part of an abstract-syntax tree.

To test for element-consistency, create minimal instances for each meta-model element both for classes *and* for properties, so references and attributes. Note that even if all classes can be instantiated, this does not mean that all properties can be populated. This requires an additional check. Each of the minimal instances should start with the root meta-model instance (the model, the document root, and so on) and add the minimal amount of other elements to show instantiation for some target meta-class or property. Obviously, instances created when testing requirements already prove consistency for many elements, so you only need to add instances for elements that have not been covered so far. Also, because many model elements require a substantial number of parent classes, you will need many fewer minimal instances than the number of meta-model elements. If your meta-model is modularized, you can also create the test cases separately for each module, as they are likely to be tested separately later (for instance you might be testing an expression sub-language implementation, separately from the rest of the abstract syntax). For small meta-models, this is usually not necessary. As before, remember to store all the created instances for future use.

Create a *maximal example* that instantiates all elements that you believe should be possible to instantiate together in a single model. In general, there is no guarantee that a single maximal example exists for every meta-model, as some meta-classes often cannot be instantiated simultaneously, but even if this is the case, it is useful to approximate it and create a large example, or a small number of such. This maximal example(s) will constitute a very practical test case for implementation of the static and dynamic semantics in the other language development activities. Save them together with other test cases created.

Now, if you created new instances, this means that some elements have not been directly traced to requirements. This might be a sign that your meta-model is not minimal (cf. *Avoid scope creep*, p. 64).

Definition 3.5. *A meta-model is* parsimonious *if it contains no meta-classes, no relations (references, associations), and no attributes that do not address any system requirements for the modeling language.*

To ensure parsimony, we recommend a systematic review of all meta-model elements (classes, attributes, relations) with respect to the language requirements. Elements introduced overly zealously by the designers should be removed (or requirements adjusted if they are justified). Finally, we recommend investigating other bad smells in the design, particularly those listed in Sections 3.4 and 3.6.

The testing process for abstract syntax and meta-models is largely independent of whether we use a meta-modeling technology (such as class diagrams and Ecore) or a type-modeling technology (such as algebraic data types, ADTs). Inconsistency problems are less likely in ADT modeling, but still possible. All the other issues apply to both styles. What mostly changes are the formats in which the instances are saved.

> **Exercise 3.11.** Describe how would you validate the meta-model presented in Fig. 3.1, i.e., explain how would you make sure that the design is satisfactory. What test cases and how many would you use? What are the main properties you want to test a meta-model for?

3.9 The Language-Conformance Hierarchy

Now we know how to describe the abstract syntax of languages using class diagrams, and since we will do that in a language workbench, let us look at the typical architectures of such workbenches. The meta-modeling hierarchy describes the common architecture that all language workbenches share. As such, its main purpose is to provide a framework that helps developers of language workbenches to design and implement them.

Imagine that you are the developer of a new language workbench. The workbench needs to provide the language engineer with some means to create a meta-model, so it offers class diagrams, Ecore, MOF, ADTs, or another suitable meta-modeling language. You chose Ecore, so you need to implement a tool that your language engineers can use to create an Ecore model. Your users create the Ecore models with the tool and need to generate the language infrastructure to allow their users to instantiate the language models. The language workbench needs to make sure that this infrastructure supports only the creation of valid models. Then, finally, in a running system, which is either an interpreter or a code generator, the model is loaded in main memory and will be traversed there. So, what we have is a hierarchy of models at different levels of abstraction, and the models are related via instantiation—a *conformance* relation.

Even the very top level, Ecore, is a model itself. While the specification of Ecore is usually only implicit in the language workbench, it turns out that one can (retro-actively) provide an Ecore model representing Ecore's abstract syntax; likewise, one can provide a MOF model representing MOF's syntax, as well as a UML model representing UML's syntax. One can even build an Ecore meta-model for the abstract syntax of Scala, or write Scala ADTs for the abstract syntax of Ecore itself. When we start to talk about meta-modeling of meta-modeling languages we end up with a hierarchy of models at different levels of abstraction. This hierarchy is called the meta-modeling hierarchy, or the *language-conformance hierarchy*.

The Meta-Model of Ecore

Let us first take a look at the top of the hierarchy, which in the case of many Eclipse-based language workbenches, is an Ecore model. We will define (and draw) this model using Ecore itself. We call this model the Ecore meta-model, since it defines the Ecore language. Thereafter, we will take a look at the hierarchy below the Ecore meta-model. Remember that we instantiate it to define our own language, such as the `robot`, the `mindmap`, or the `fsm` DSLs. These models are then meta-models themselves, since they define all possible instances (models) in our DSL (e.g., the random walk program from Figures 2.2 and 2.5 written in `robot`). Given this hierarchy, from the perspective of models or programs that are instances of our language, the Ecore meta-model is therefore also often called a meta-meta model.

Figure 3.12 shows an excerpt of the Ecore meta-model, which is expressed in the Ecore language itself. In other words, the concrete syntax of the Ecore language is that of class diagrams, so we use this notation to draw the Ecore meta-model. The full Ecore meta-model has more than 50 classes. In the figure, we only show the core classes and their relationships; we also hide many of the attributes and all operations.

As you can see, when instantiating the Ecore meta-model in your own model, you can use well-known class-modeling constructs. For instance, use EClass to represent classes in your model, add attributes (by instantiating EAttribute) or relationships (by instantiating EReference) to it, and organize your classes in a package hierarchy (by instantiating EPackage). EReference is a good example of a reified relationship (cf. Sect. 3.4), since relationships between classes in a model have properties, such as whether the relationship represents a containment (cf. attribute EReference.containment).

Interestingly, many methods are defined in the Ecore meta-model, in an apparent contradiction to what we recommended in Sect. 3.4: that meta-models should not contain operations. Some of these methods realize derived properties, such as the method EClassifier.getClassifierID. However, most of these methods belong to the reflective API of Ecore that can be used when no Java classes are generated from a meta-model. These are to be used by reflective tools that operate on arbitrary meta-models. In fact, Ecore can be used completely without using code generation. To this end, an Ecore model (as a meta-model for a DSL) can be created dynamically at runtime, instantiated, and then processed (e.g., traversed or modified) using the reflective Ecore API. Such a runtime instance of an Ecore language is called a dynamic instance, further explained in the appendix “Using the Eclipse Modeling Framework” on our book website (http://dsl.design).

The Ecore meta-model of Fig. 3.12 has been created post factum, after Ecore was already implemented. Of course, the language has to be implemented before it can be used to describe models (i.e., other languages) in it.

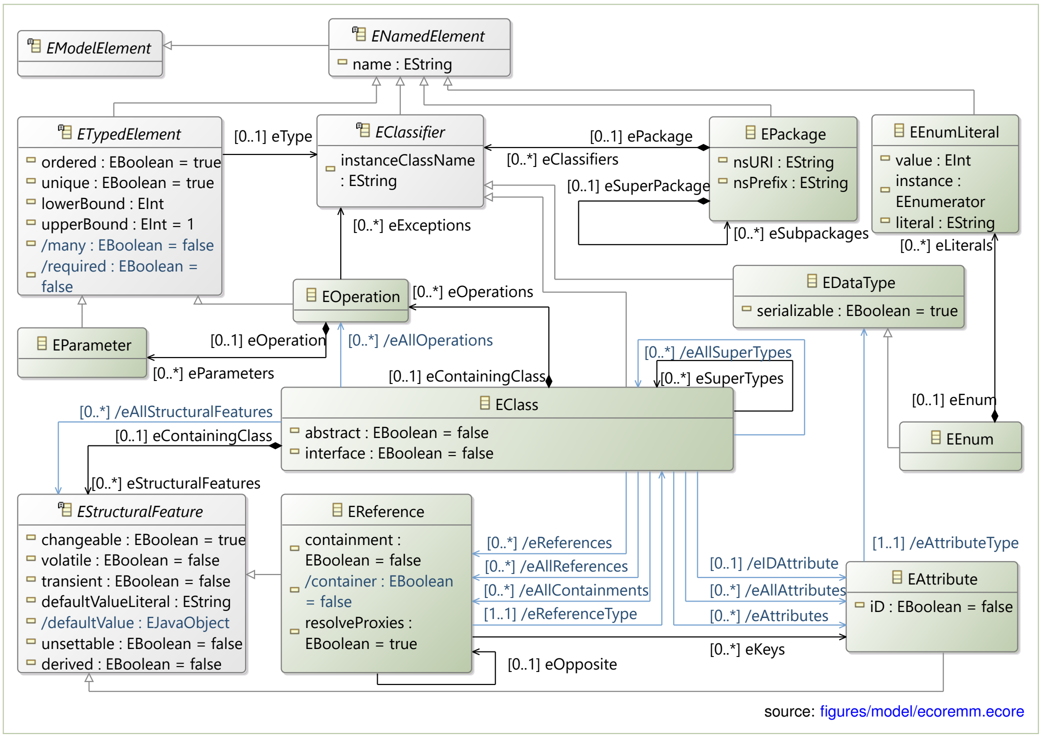


Figure 3.12: An excerpt of the Ecore meta-model of Ecore. In other words: the meta-model of the Ecore language, where the meta-model is expressed in Ecore itself

So, the EMF developers first implemented Ecore and then defined the Ecore meta-model for it using the language. This method is called *bootstrapping*, and originates in compiler construction.

Bootstrapping: Describing a Language in Itself

The fact that one can describe the abstract syntax of a class-modeling language, such as Ecore, using class modeling itself has actually sometimes led to confusion that some languages are defined in themselves, for example 'UML defined in UML' or 'Ecore is defined in Ecore.' Such statements are false—circular definitions of languages are not possible.

The practice of modeling a language using itself could better be called *bootstrapping*. It resembles a practice common among programming language designers, who build compilers for a new language in the language itself, as the first serious maturity test. For example, your favorite Java compiler is most likely implemented in Java. Of course, a bootstrapped language first needs a compiler or an interpreter implemented in another language (which already has a compiler or interpreter). Typically one first implements an interpreter for the core language (say Java) in a language with an existing compiler (say C). Once this implementation works, one reimplements the interpreter/compiler in Java again, and throws away the temporary C-based

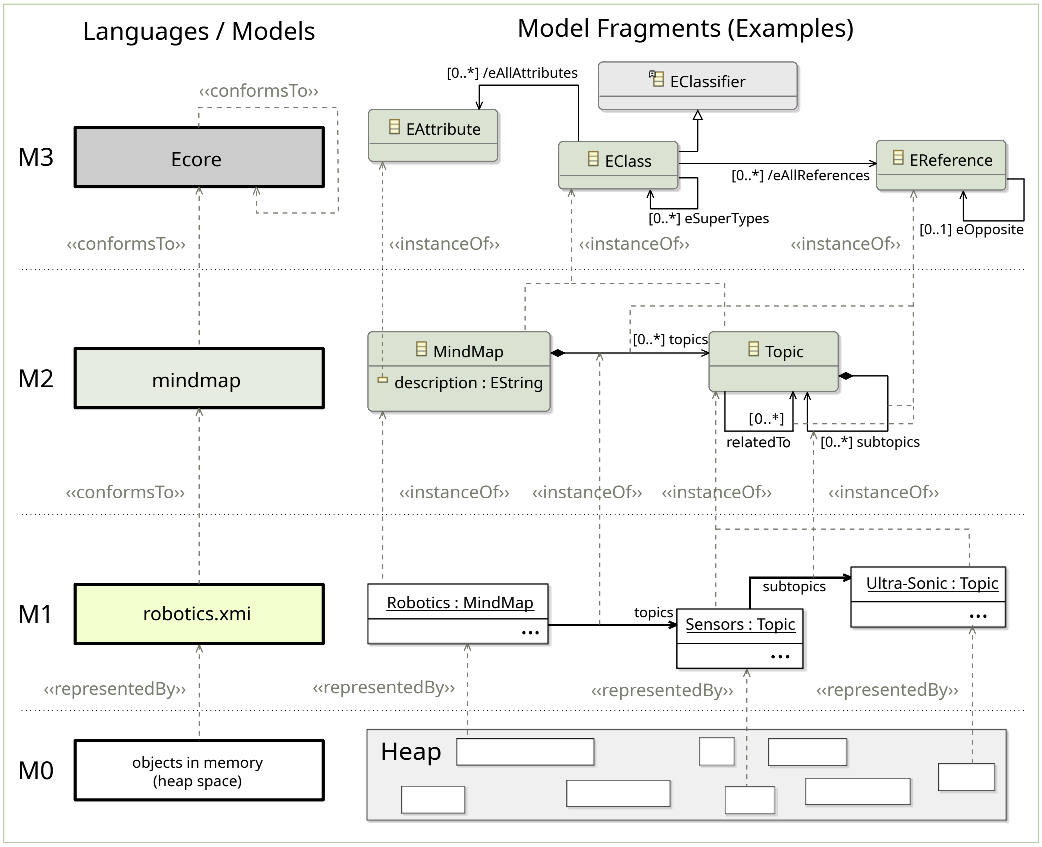


Figure 3.13: The conformance hierarchy for Ecore and the `mindmap` language

interpreter. Similarly for modeling languages: the first definition uses an existing language, or simply a natural-language description. The bootstrap-like self-definition comes later, once the modeling language already exists.

Meta-Modeling Levels

The Object Management Group organizes models and languages in a hierarchy of abstraction layers, also known as the MOF meta-modeling hierarchy. This is exemplified in Fig. 3.13 using our mind-map language and Ecore as the meta-modeling language. Recall that Ecore is the de facto reference implementation of MOF.

In the figure, at the very top level, called *M3*, we have the Ecore language, which allows description of class diagrams. Instances of this language are class diagrams at the level *M2*. A class diagram describing an abstract syntax (the meta-model) of the mind-map language belongs here. Note the *conformsTo* relation between languages and models, and we use the *instanceOf* relations between model elements. On M3, the Ecore language *conformsTo* itself. On M2, our mind-map DSL *conformsTo* Ecore.

One level below at *M1*, we have a concrete model in the mind-map language, here shown using a notation of instance specifications, sometimes referred to as object diagrams (so their abstract syntax is shown). The models at *M1* describe concrete mind-map notes; here, a mind map of some robotics topics (sensors). It *conformsTo* our mind-map language.

At the bottom level of the hierarchy, *M0*, we have the real system, specifically, the objects that exist in the main memory at runtime. These objects are *representedBy* the models at *M1*, and these are the objects you will traverse and process programmatically (as defined by your dynamic semantics). For instance, when you write an interpreter, you will traverse these objects; or, when you write a generator or a code transformation, these objects will be the actual input.

Some authors instead say that *M0* refers to the 'physical world' (or the domain) that is represented by the models at *M1*. One could see it like that, but we think that this can be confusing, especially since there is not always a physical world that your model will represent. For instance, what exactly would a mind-map topic "Sensors" represent? Instead, always keep in mind that at some point there need to exist real objects in the computer memory that can be traversed and processed in some way.

Now, for the model layers M1–M3, let us briefly discuss what kind of syntax is shown. On levels M3 and M2, we use the concrete syntax of Ecore to show the excerpts of the models—exactly the same way an Ecore model would be shown in the graphical editor available in the Eclipse Modeling Framework. On M1, we use the abstract syntax of our mind-map language, so we show the model as an object diagram. If we had used a concrete syntax, it could look like Fig. 3.8 or Fig. 3.9, depending on how we had designed the concrete syntax.

Figure 3.14 shows the same architecture, but using UML instead of Ecore to define the mind-map language. The design of the UML has actually been the main rationale for organizing this architecture. To formally define UML, the MOF language was created by OMG. Recall that MOF, very similar to Ecore, is a very simple class-modeling language that is less expressive than UML class diagrams. For this reason, it is very usable to define the abstract syntax of languages, including that of the very complex UML language with its different sub-languages (class diagram, sequence diagram, state-machine diagram, etc.). Since then, it has proven very useful for understanding the layers involved when designing DSLs, like our mind-map language. To understand the remainder, note that the models in M1–M3 are all shown in concrete syntax, as opposed to Fig. 3.13, where M1 was in abstract syntax for convenience reasons.

The UML hierarchy is a bit tricky, though. While on M3, MOF *conformsTo* itself, the entire UML language is in *M2* and *conformsTo* MOF. Importantly, UML contains conformance of models to meta-models in itself. So, UML allows you to model both class diagrams and their instances (i.e., object diagrams) in the same model. This appears to be in conflict with the

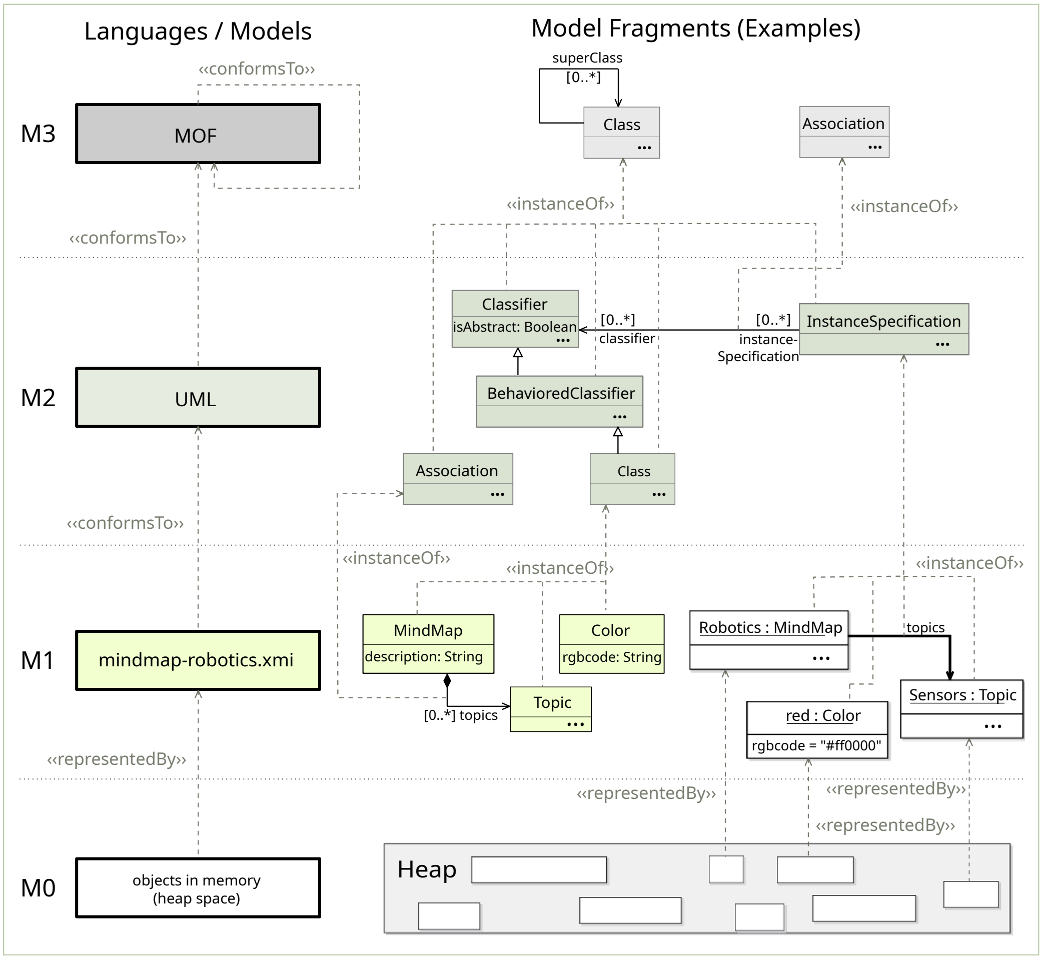
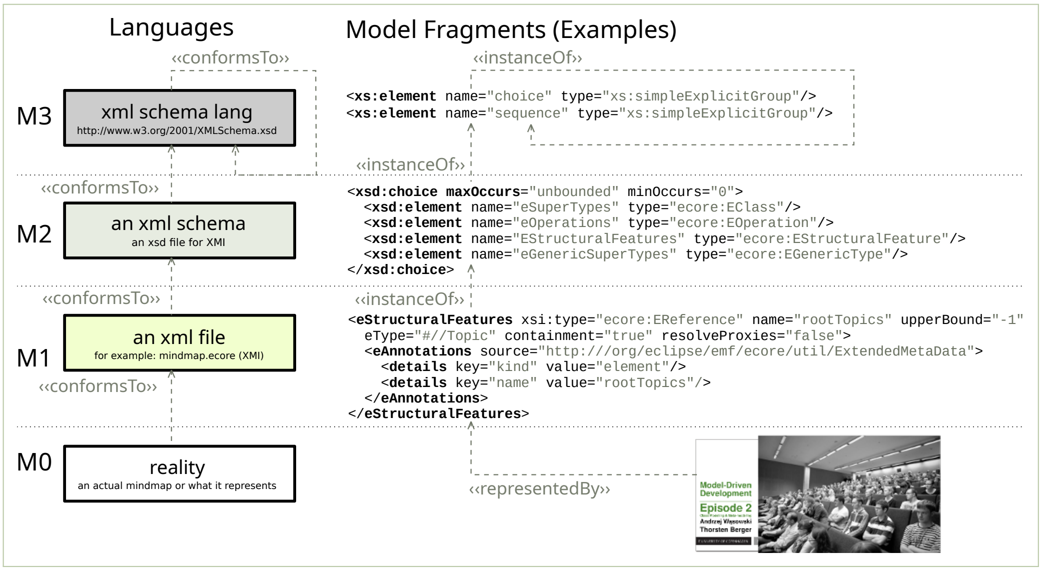


Figure 3.14: Meta-modeling hierarchy illustrated using UML and the `mindmap` *language*

meta-modeling hierarchy: we have types and instances at the same level, or a language (UML) that stretches over two levels. In the example, we can put both our mind-map DSL and their instances into level M1. You can see this as a kind of unification of classes and their instances, which has advantages that we discuss shortly. The figure actually shows how UML tools are implemented to support this, using a general language-processing stack, such as EMF, and this stack is at the same abstraction level as M1. The key idea is that UML models the *instanceOf* relation itself, in the language, while EMF just implements it in its language-processing stack. In other words, the UML *instanceOf* relation between a class and an instance (i.e., an object) becomes a regular reference (association) in the implementation.

Using this support, the so-called *ontological instantiation* (explained shortly), you can use UML to model classes in M1, which are instances of the UML class Class in M2, and you can model instances (i.e., objects)



```
<xs:element name="choice" type="xs:simpleExplicitGroup"/>
<xs:element name="sequence" type="xs:simpleExplicitGroup"/>
```

```
<xsd:choice maxOccurs="unbounded" minOccurs="0">
  <xsd:element name="eSuperTypes" type="ecore:EClass"/>
  <xsd:element name="eOperations" type="ecore:EOperation"/>
  <xsd:element name="EStructuralFeatures" type="ecore:EStructuralFeature"/>
  <xsd:element name="eGenericSuperTypes" type="ecore:EGenericType"/>
</xsd:choice>
```

```
<eStructuralFeatures xsi:type="ecore:EReference" name="rootTopics" upperBound="-1"
  eType="#//Topic" containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="element"/>
    <details key="name" value="rootTopics"/>
  </eAnnotations>
</eStructuralFeatures>
```

Figure 3.15: *Meta-modeling hierarchy illustrated using the XML technology stack*

in M1, which are instances of the UML class InstanceSpecification. On M2, both these classes are associated to each other; see the arrow labeled classifier and instanceSpecification.

In M1, let us take a look at the instances on the right-hand side. Since the model is shown in concrete syntax, what you can see is three objects connected by two links. All of these are actually represented by five objects in the abstract syntax. If we were to show the abstract syntax, you would see the five objects. However, the UML specification defines the respective concrete syntax as follows: if an object is an InstanceSpecification whose classifier is a Class, then the object is rendered in the typical object notation. If an object is an InstanceSpecification whose classifier is an Association, then it is rendered as a link (an arrow that has a label).

In summary, the idea of the meta-modeling hierarchy is that various levels of meta-modeling can be set at various abstraction levels. For example, a meta-model of Ecore is very abstract. A meta-model of UML expressed in MOF is more concrete. A model of a mind-map application expressed in UML is even more concrete. An instance of that model, an actual mind map expressing specific topics, is very concrete.

When you design your own DSL using Ecore, it belongs to level *M2*, replacing UML, and concrete models in the DSL are at level *M1*. It depends on the concrete project whether they have further instances or not—usually not. If you design your DSL using UML, you will most likely use one more layer.

Linguistic Versus Ontological Instantiation

We have seen how EMF and UML support the instantiation of meta-models. In EMF, the instance model is always a different model, whose conformance to its meta-model is assured via the language-processing stack in EMF. In UML, you can create a meta-model and (parts of) its instances in the same model, since UML supports conformance as part of its language. The former is called a *linguistic*, and the latter an *ontological* instantiation [1], also known as multi-level modeling [6, 18]. The different approaches in their whole are also referred to as linguistic meta-modeling and ontological meta-modeling [15]. The latter is inspired by typical ontology specification languages, such as OWL, which support modeling both classes (i.e., types) and their instances, called TBox and ABox statements, respectively [3].

Now, what is the benefit of that? Conceptually, we are bringing classes and their instances to the same level of abstraction. Often, when you program (or model) you think about the program (or model) and manipulated values (instances) simultaneously, so why not model them together? Programming languages allow both algorithms and values to be specified. This duality is often needed in modeling DSLs as well. One common use case is to provide a collection of predefined "runtime" objects in a language.

A useful example can be found in the M1 layer for our mind-map example in Fig. 3.14. Ignore the objects Robotics and Sensors and suppose there would only be the object red. Remember that above in Sect. 3.7, we lamented about the missing possibility to predefine some concrete colors for our mind-map language, which is not possible in Ecore. In UML, you can just create some instance specifications for the colors that you want. Instantiate them with the respective color codes as attribute values. This way, you define that in the system at runtime these instances need to exist. You can of course still separate instance models and the definition of our mind-map DSL, but conceptually, these models reside on the same meta-level, M1. It is unified in a way that you can just import your meta-model; you could extend the meta-model or partially instantiate it. This allows more expressive DSLs to be designed [6, 18, 1, 15, 2].

3.10 A Sneak Peek at XML

We have discussed the meta-modeling hierarchy as established by MOF and realized using class diagrams from UML and EMF's Ecore language. You are probably familiar with XML together with its related technologies, such as XSD, XSL, XSL-FO, and so on. These technologies also form a meta-modeling hierarchy, similar to the one we explained above.

The example in Fig. 3.15 shows the same meta-modeling architecture as realized by the W3C technology stack for structured data XML. At the top level, *M3*, we have the XML Schema Language XSD, which conforms to its specification in XSD—again, after the language has been designed. It has been described in itself and the corresponding XSD file has been published.[6] At the *M2* level we have XML Schemas for concrete languages. Here, we use the XMI language as an example. At the *M1* level we have

[6] http://www.w3.org/2001/XMLSchema.xsd, retrieved 2022/09

concrete XML files conforming to the schema of *M2*. In the example, we use the mindmap.ecore file that conforms to the XMI schema for model representation in XML format.

The familiar XML stack has very similar aims to meta-modeling languages: describing structures and data in a standard manner. The main differences are that (1) XML documents are not really meant to be processed by humans, and that (2) XML processing stays largely on the level of strings or trees. Tools for processing models usually stay at a higher abstraction level. As we will see in later chapters, models are processed using languages that support standard object-oriented programming models.

Further Reading

Fowler and Parsons [8] distinguish domain-models and meta-models, and, like us, they strongly argue for the use of explicit meta-models in the design and implementation of DSLs. They use the name *semantic models* for meta-models and they devote an entire section to the pattern of using them in language implementations.

The community advocating MDSE and meta-modeling is commonly known as the *modelware* community. However, as we indicated at various opportunities, language development is an old discipline and traditionally centers around grammars and parsers. That community is known as the *grammarware* community. Grammarware practitioners might find the concepts of modelware as described in this book unusual. Paige, Kolovos, and Polack [20] provide an introduction into meta-modeling concepts for grammarware practitioners. Among other things, they motivate the use of meta-models that are still (relatively) unknown in the grammarware community.

The Meta-modeling hierarchy is described in section 6.2 of the book by Stahl and Völter [21]. This hierarchy can also be found in the *UML 2.5 Infrastructure Specification* from the Object Management Group.

We used the Eclipse Modeling Framework EMF with its language Ecore to illustrate meta-modeling. Beyond the quick tutorial on EMF available on the book's website, we recommend the following books of Steinberg et al. [22], Budinsky et al. [5], and Moore et al. [16] to for in-depth information about EMF.

Additional Exercises

Exercise 3.12. Specify an object diagram (a class instance specification diagram) presenting the abstract-syntax tree of the mind map shown in Fig. 3.9. Limit your diagram to the root topic (“Class Modeling, Meta-modeling”), the sub-topic “3. Meta-Modeling,” and two of its sub-topics. Use the meta-model seen in Fig. 3.10.

Exercise 3.13. The object diagram in Fig. 3.16 shows (supposedly) an instance of the meta-model of Fig. 3.1. What is the problem with this instance? Could it represent a legal syntax tree? Why? **a)** Find a conformance error in this instance. **b)** Correct the object diagram so that it conforms to the meta-model.

Exercise 3.14. Load the mind-map meta-model into a modeling tool and create an instance of "Document Root" representing the abstract syntax of the model shown in Fig. 3.9. For the purpose of the exercise, assume that topics are represented by blue boxes in the concrete syntax. Threads are represented by white boxes with a little blue circle. Thread items are represented by lines branching out of

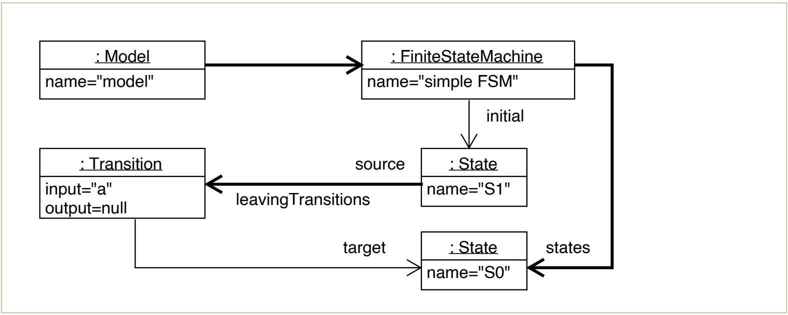


Figure 3.16: An example invalid instance of the `fsm` meta-model of Fig. 3.1

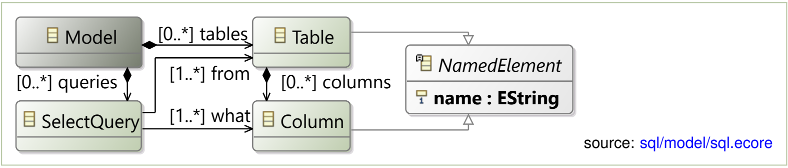


Figure 3.17: A simple meta-model for SQL queries

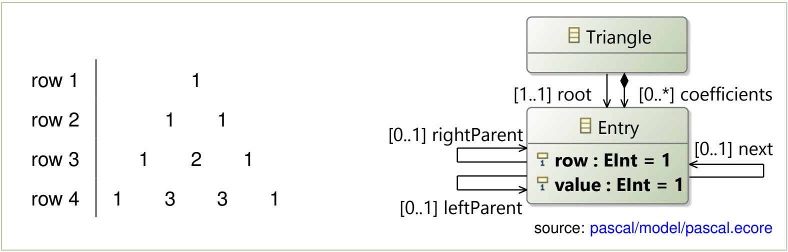


Figure 3.18: A meta-model for Pascal's triangle

the thread's blue circle. The mind-map meta-model in Ecore format is available from the mindmap project of the book repository.

Exercise 3.15. Figure 3.17 presents a simplified meta-model for SQL queries. Draw object instance diagrams representing the abstract-syntax trees of the following queries as instances of this meta-model. You will need to invent a suitable data model with tables and columns.

a) `SELECT NAME FROM CUSTOMER;`

b) `SELECT NAME, PRICE FROM PRODUCT;`

Exercise 3.16. Pascal's triangle (Fig. 3.18) is a numeric hierarchical structure, where each internal node's value is the sum of the values of its two parents, a row above. The right part of Fig. 3.18 presents a meta-model to represent Pascal's triangles of different sizes. Numbers are stored in entries nested directly under an instance of the root class `Triangle`. Additional references connect nodes to parents and to their next sequential neighbor. Draw the abstract syntax of the triangle in the left part of the figure as an instance of the meta-model in the right side. To save time, draw the instance for the first three rows (ignore row 4).

Exercise 3.17. Figure 3.19 shows a feature model in concrete syntax, and Fig. 3.20 shows a simplified meta-model for this language. In concrete syntax, we draw a hollow arc to denote `XorGroup` and a filled one to denote `OrGroup` members. There is only an Xor group in the instance. The feature `aircon` is an optional feature,

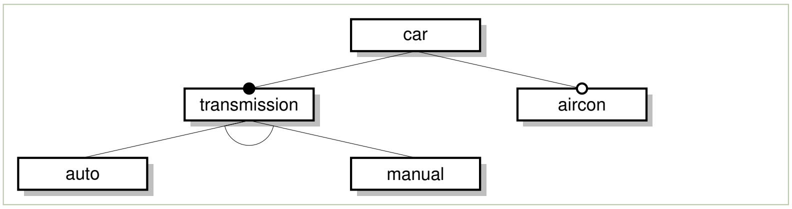
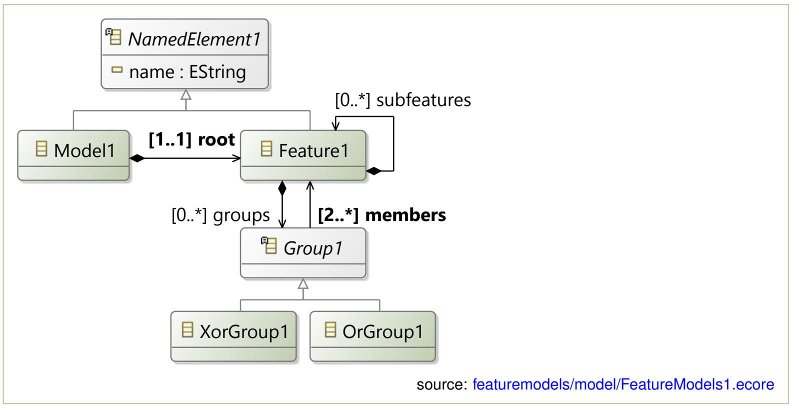
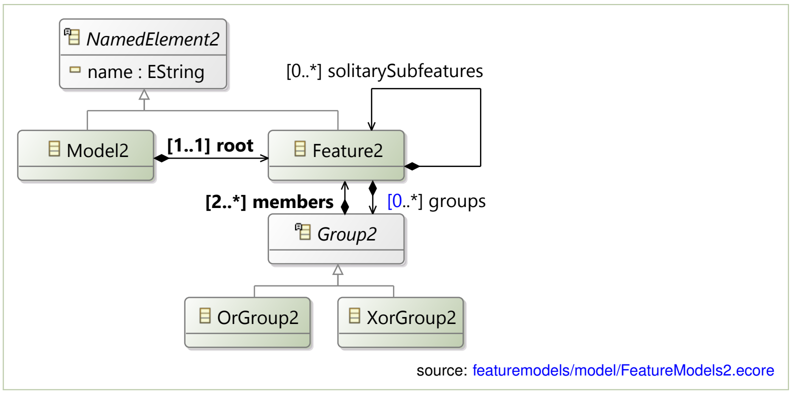


Figure 3.19: A simple feature model in concrete syntax



source: featuremodels/model/FeatureModels1.ecore

Figure 3.20: A meta-model for feature diagrams



source: featuremodels/model/FeatureModels2.ecore

Figure 3.21: An alternative meta-model for feature-models

denoted by the hollow circle, whereas the feature `transmission` is mandatory, denoted by the filled circle. Draw the abstract syntax of the above feature model as an instance of this meta-model. Remember that an abstract-syntax tree must have a single partonomy. Can the optionality of a feature be represented in the meta-model? More information about feature models is available in Chapters 11 and 12.

Exercise 3.18. Draw the abstract syntax of the feature model of Fig. 3.19, as an instance of an alternative meta-model for feature diagrams shown in Fig. 3.21.

Exercise 3.19. Extend the meta-model in Fig. 3.20 so that it supports *excludes* and *requires* constraints. After the extension it should be possible to state in the syntax of the modeling language that some feature requires another feature, or

```
vertex 1;
vertex 2;
vertex 3;
edge 1->2 [coin];
edge 2->4 [coffee];
edge 3->1 [deliver]
```

Figure 3.22: *An example graph in a hypothetical concrete syntax*

```
abstract class HasOptions {};
abstract class NamedElement {};
abstract class Question extends NamedElement {};
class MultipleChoice extends Question, HasOptions {};
class SingleChoice extends Question, HasOptions {};
```

Figure 3.23: *An example model in a hypothetical concrete syntax*

some feature excludes the use of another feature. For instance:

electric *requires* automaticTransmission
diesel *excludes* hybrid

Exercise 3.20. The meta-model of Fig. 3.21 does not allow optional features to be represented. Fix the meta-model by modifying the diagram so that it can represent the distinction between optional and mandatory features.

Exercise 3.21. An HTML document consists of a header and a body. The header has an attribute 'title' of type string. The body is a nested tree of elements and text chunks containing strings of characters. Elements can nest other elements and other chunks of text underneath. Text chunks cannot nest anything. Only two types of elements are allowed: paragraphs (`p`) and divisions (`div`). Design a meta-model representing documents in this subset of HTML. Include anchors and anchor references in the meta-model. Use either Ecore or ADTs for modeling. If you do both, compare the differences and similarities.

Exercise 3.22. Consider a simple modeling language for describing labeled directed graphs. An example in concrete syntax is shown in Fig. 3.22. A graph consists of a number of vertex declarations, each naming a vertex with an integer number, and a list of edge declarations, each relating to vertices with an optional label (a character string). Design an Ecore meta-model (or ADTs in your functional programming language of choice) for representing such models.

Exercise 3.23. Consider a simplified variant of the Google Protocol Buffers DSL (Sect. 1.1). We use a simplified language in this task: a model consists of a number of message types. Each message type has a number of attributes and a name. Each attribute has a name and a type (another message type), and a Boolean attribute specifying whether the attribute is optional or mandatory. Present an Ecore meta-model, or a set of ADT definitions, for the language described above.

Exercise 3.24. We want to describe very simple class models. Each class has a name; it can be abstract or concrete; and a class may extend several other classes (multiple inheritance). Figure 3.23 shows an example model in concrete syntax. Design an Ecore meta-model able to represent abstract syntax of such models.

Exercise 3.25. Describe the AST of the language of the previous exercise using an ADT instead of class diagrams (use a suitable language like Haskell, F#, or Scala).

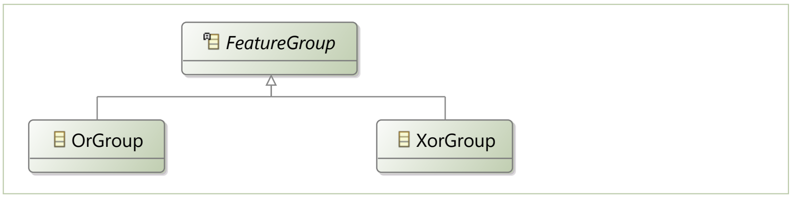


Figure 3.24: *A simple class diagram containing a core part of a meta-model for feature diagrams*

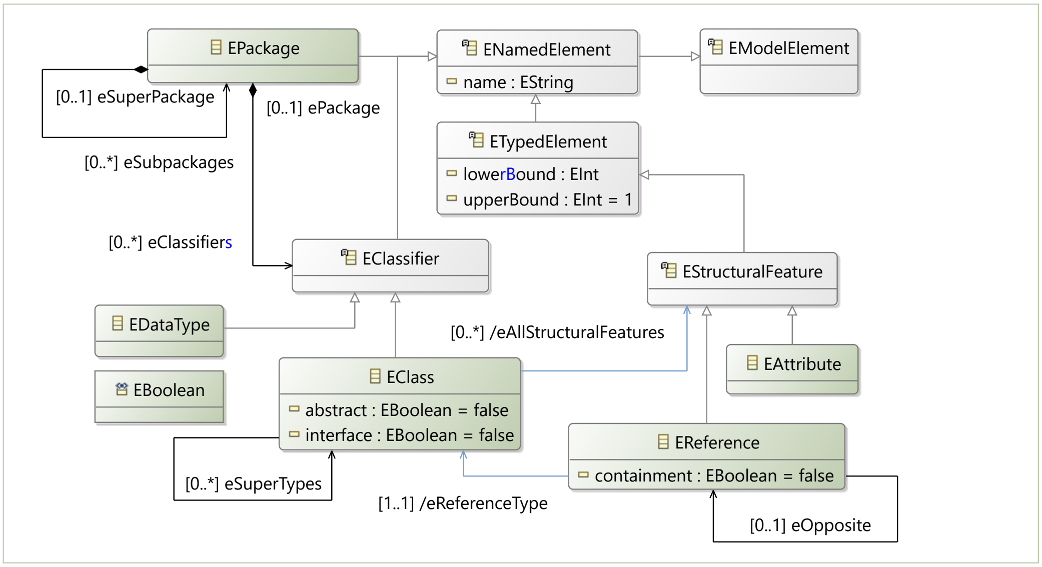


Figure 3.25: *A subset of the Ecore meta-model*

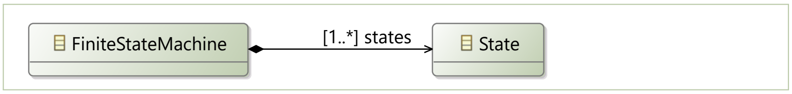


Figure 3.26: *An over-simplified meta-model for state machines*

Exercise 3.26. A simplified XML document is a tree of elements. Each element has a name, a list of parameters, and a list of nested elements. Each parameter has a name and a value of type string. Design a meta-model representing XML documents from this simple dialect. Note that this is meant to be done for XML as a language, not for a particular XML dialect.

Exercise 3.27. Design a meta-model for an XML dialect known to you. Explain the main difference between this meta-model and the one in the previous exercise.

Exercise 3.28. Draw (on paper) the partonomy and taxonomy views for the meta-model of Ecore (Fig. 3.12).

Exercise 3.29. Figure 3.24 shows a feature-diagrams' meta-model. Draw the abstract-syntax tree of this diagram as an instance of the meta-model in Fig. 3.25.

Exercise 3.30. Figure 3.26 depicts a simple Ecore class diagram (incidentally) describing a fragment of a meta-model for state machines. Draw the abstract-syntax tree of this diagram as an instance of the Ecore meta-model shown in Fig. 3.25.

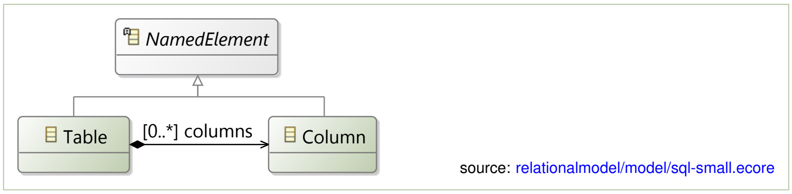


Figure 3.27: *A tiny meta-model for relational data (entity-relationship diagrams)*

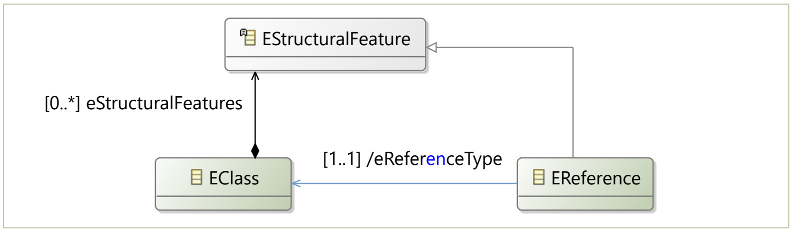


Figure 3.28: *A fragment of the Ecore meta-model*

Exercise 3.31. Figure 3.27 shows a meta-model for relational schemas. Draw the abstract syntax of this diagram as an instance of the meta-model in Fig. 3.12. Include classes, generalizations, references, and properties such as abstract, containment, cardinality constraints (upper and lower bound), and names.

Exercise 3.32. The meta-model of Ecore shown in Fig. 3.25 is itself an Ecore model. Thus it can be presented as an instance of itself, a kind of boot-strapping. We attempt to understand this idea on a small part of Fig. 3.25 shown in Fig. 3.28. Draw the abstract syntax of the diagram in Fig. 3.28 as an instance of the Ecore meta-model in Fig. 3.25. Include classes, generalizations, references, and properties such as abstract, containment, and names.

Exercise 3.33. Ecore supports bidirectional associations only indirectly (see section "Associations" in the appendix "Class Modeling" on our book website, http://dsl.design). It uses the EOpposite property to relate two inverse unidirectional references. Study the Ecore meta-model (Fig. 3.12) and explain how bidirectional references are represented in abstract syntax. Draw the abstract syntax for a simple object diagram showing two classes related by a bidirectional reference.

Exercise 3.34. This exercise can be solved after reading Chapter 5. Use your favorite meta-modeling mechanism to design a meta-model for Alloy instances, like those shown in Fig. 5.14 on p. 173. See also Exercise 4.57 on p. 141.

References

[1] Colin Atkinson and Thomas Kühne. "Model-driven development: a meta-modeling foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41 (cit. p. 78).
[2] Colin Atkinson and Thomas Kühne. "Reducing accidental complexity in domain models". In: *Software & Systems Modeling* 7.3 (2008), pp. 345–359 (cit. p. 78).
[3] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003 (cit. p. 78).
[4] Jon Bentley. "Programming pearls: Little languages". In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721 (cit. pp. 65, 68).

[5] Frank Budinsky, David Steinber, Ed Merks, Raymond Ellersick, and Timothy Groose J. *Eclipse Modeling Framework*. Addison-Wesley, 2004 (cit. p. 79).

[6] Victorio A. Carvalho and João Paulo A. Almeida. "Toward a well-founded theory for multi-level conceptual modeling". In: *Software & Systems Modeling* 17 (2018), pp. 205–231 (cit. p. 78).

[7] Johannes Ernst. *What is metamodeling and what is it good for?* Unfortunately, the original website is no longer available. 2002 (cit. p. 47).

[8] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley, 2011 (cit. pp. 59, 79).

[9] Ralf Gitzel and Tobias Hildenbrand. *A Taxonomy of Metamodel Hierarchies*. Apr. 2005. URL: https://madoc.bib.uni-mannheim.de/993/ (cit. p. 48).

[10] Kyo Chul Kang. "FODA: Twenty years of perspective on feature models". In: *SPLC*. Keynote Address. 2009 (cit. p. 48).

[11] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. "Design guidelines for domain specific languages". In: *9th OOPSLA Workshop on Domain-Specific Modeling*. 2009 (cit. pp. 64, 65).

[12] Steven Kelly and Risto Pohjonen. "Worst practices for domain-specific modeling". In: *IEEE Software* 26.4 (2009), pp. 22–29 (cit. pp. 64, 65).

[13] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: A framework for compositional development of domain specific languages". In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.5 (2010), pp. 353–372 (cit. p. 65).

[14] Glenn E. Krasner, Stephen T. Pope, et al. "A description of the model-view-controller user interface paradigm in the Smalltalk-80 system". In: *Journal of Object-Oriented Programming* 1.3 (1988), pp. 26–49 (cit. p. 57).

[15] Alfons Laarman and Ivan Kurtev. "Ontological metamodeling with explicit instantiation". In: *SLE*. 2009 (cit. p. 78).

[16] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM, 2004 (cit. p. 79).

[17] Damir Nesic, Jacob Krueger, Stefan Stanciulescu, and Thorsten Berger. "Principles of feature modeling". In: *FSE*. 2019 (cit. p. 48).

[18] Bernd Neumayr, Katharina Grün, and Michael Schrefl. "Multi-level domain modeling with m-objects and m-relationships". In: *6th Asia-Pacific Conference on Conceptual Modeling*. APCCM. 2009 (cit. p. 78).

[19] Object Management Group. *Metadata Interchange (XMI) Specification*. 2015. URL: https://www.omg.org/spec/XMI (cit. p. 48).

[20] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. "Metamodelling for grammarware researchers". In: *5th International Conference on Software Language Engineering (SLE)*. 2013 (cit. pp. 65, 79).

[21] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005 (cit. pp. 65, 79).

[22] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. 2nd Edition. Addison-Wesley, 2009 (cit. p. 79).

[23] Wisława Szymborska. *Enough*. 2011 (cit. p. 47).

[24] David S. Wile. "Lessons learned from real DSL experiments". In: *Sci. Comput. Program.* 51.3 (2004), pp. 265–290 (cit. pp. 50, 64, 65).

Parser development
is still a black art.
Klint, Lämmel, and Verhoef [16]

4 Concrete Syntax

Models and meta-models, algebraic data types and values, XML schemas and files, class and instance diagrams, YAML files—all these abstract-syntax specification methods are clearly important for you as a language designer. At the same time, the end-users, especially domain experts who are not programmers, tend to find them unnatural and cumbersome to use. An important part of a domain-specific language design is to choose a natural and easy to use *concrete syntax*, so that users can work efficiently.

In this chapter, we define what is concrete syntax, and detail how to create syntax that is easy to read and write for language users, and that is understandable and maintainable for the language designers. We discuss specification mechanisms (context-free grammars and regular expressions), design guidelines for textual syntax, and quality assurance. However, we try to limit the theoretical considerations to a bare minimum. We hope that with this chapter we can actually meet the wishes of Klint, Lämmel, and Verhoef [16] expressed in the paper quoted at the top of this page: to demystify creation of parsers, showing and systematizing how grammars are written. The chapter contains examples, case studies, and exercises, but also many practical rules and guidelines on how to arrive at a good design of concrete syntax, expressed in a robust grammar.

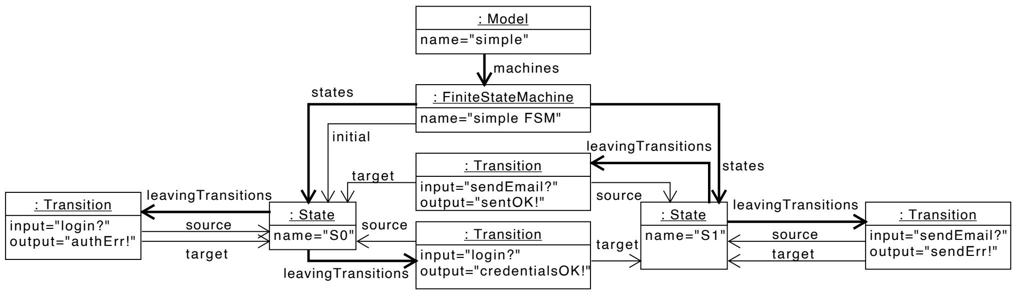
4.1 Concrete and Abstract Syntax

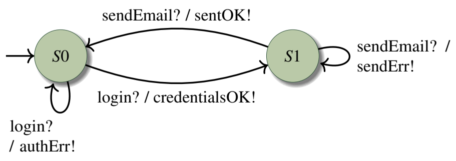
Figure 4.1 shows a model of a finite-state machine in three different representations. The well-known *graphical concrete syntax* is found in the bottom left, repeated from Tbl. 3.1. Meant for human consumption, it uses graphical elements (e.g., arrows) and characters to represent the model. An object-oriented *abstract syntax* of the very same model is shown in the top, repeated from Fig. 3.3. Finally, the bottom-right part of the figure shows the very same model in a *textual concrete syntax*, a text-based representation aimed at human readers. Textual representations tend to be the easiest user-oriented representations to define and implement. Users (engineers) prefer them over graphical ones if large models need to be created or inspected manually. This chapter focuses on defining such textual concrete syntax, and on parsing it to obtain the corresponding abstract syntax.

Let us state explicitly the definition suggested above.

Definition 4.1. Concrete syntax *is a representation of the model that is seen, produced, and manipulated by the language user. Concrete syntax is called graphical if it uses drawn elements (typically lines, arrows, geometrical*

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_4





```
machine "simple FSM" [
  initial S0
  state S0 [
    on input "login" output "credentialsOK" and go to S1
    on input "login" output "authErr" and go to S0
  ]
  state S1 [
    on input "sendEmail" output "sendErr" and go to S1
    on input "sendEmail" output "sendOK" and go to S0
  ]
]
```

Figure 4.1: *A state-machine model in abstract syntax (top), concrete graphical syntax (bottom left), and concrete textual syntax (bottom right). Convince yourself that the three representations indeed capture the same model*

shapes, or icons). It is called textual if it is written as text, in a character set available in the model editor.

Why these names? Why *abstract* and why *concrete*? The abstract syntax *abstracts away* the visual aspects, including the linear or graphical layout that encodes the model structure. For example, in Fig. 4.1 the instance diagram does not contain any information on how transition arrows are routed, what is the color and size of the state ovals, and where the labels are physically placed. Nor does it store the order of transitions in the textual version. The square brackets have been replaced by association links in the instance diagram. The link labeled leavingTransitions represents the same information as the fact that an arrow is sourced in a state oval, or that an `on input` line is placed in the square bracket section of a state in the textual syntax.

A careful reader notices a paradox in Fig. 4.1: the instance diagram on top shows lots of *concrete* information, even though it presents the *abstract* syntax of a state machine. It has its own arrows, lines, boxes, and labels. This is because it is drawn in a *concrete* syntax of another language, the UML instance specification diagrams. Otherwise, you would not be able to see it! Indeed, we have no way to show the abstract syntax on paper or screen—it only exists *abstractly*, as objects and values during program execution. Whenever we want to make abstract syntax visible for human eyes, we need to write it down in some notation, using some concrete syntax. Therefore, a human-readable syntax is useful, not only if we have to write models, but also whenever models are not written by humans, created and

Instances vs Specifications for Concrete and Abstract syntax

In Sect. 3.9 we were extremely careful to specify conformance levels between elements of languages and models. We have meta-models, defining abstract syntax of all possible models in the language, and instances defining abstract syntax of a particular model. We should admit that in daily communication, and also in this book, we will often just write *abstract syntax* without specifying whether we refer to the entire language (meta-model or types) or to a specific model (objects or values). The meaning should typically be clear from the context.

processed completely automatically, but need to be read by humans for example for debugging and monitoring.

Exercise 4.1. Revisit the abstract and concrete syntax in Fig. 4.1. Explain how the following changes to the model affect each of the three representations:

a) Make state S1 initial in this machine.

b) Add a new transition from S1 to S0 with input “reset” and output “initialized.”

c) Rename state S0 to S2. Be sure that you indicate all places where the changes need to be made. How many places need to be changed in the abstract syntax? How many places need to be changed in the textual concrete syntax?

4.2 Defining Concrete Syntax

We seek a way to *recognize* concrete syntax in order to distinguish correct programs from incorrect ones, and to *translate* them into an abstract syntax, a form easy to handle for tools. To get there we need a precise and unambiguous definition of the concrete syntax, completing the loose English requirements collected during domain analysis. In order to explain how such definitions are made, we first need to put a few basic concepts on the table. This short section recalls the basic theoretical underpinnings of concrete syntax definitions, leaving the practical applications to later pages.

Lexical and syntactic structure of program text. It is useful to split the concrete syntax into two layers, traditionally called *lexical* and *syntactic*. The *lexical* structure defines what are the legal words in the language, for instance: How does a string literal look? What are the available ways to write numbers? What are the keywords and operators in the language? The *syntactic* structure defines how the words can be connected into understandable sequences, analogous to sentences in natural languages. For instance, that a variable declaration consists of a type name, followed by an identifier, and an initializer.

Definition 4.2. *The* lexical structure *determines what terms (also known as words, tokens, lexemes) are legal in a language. The* syntactic structure *defines in what order the terms (words, tokens) can appear in a model.*

Let us define the lexical structure of the `fsm` language (Fig. 4.1, bottom right) as follows:

Example 9. The keywords are: `machine`, `initial`, `state`, `on`, `input`, `output`, `and`, `go`, and `to`. String literals are any sequence of characters not containing double quotes, surrounded by double quotes. Identifiers (state names) start with a letter followed by a sequence of letters and digits. Square brackets are used to denote nesting of states and machines. White space has no meaning in this language, except that it is used to separate tokens.

The syntactic structure of this language can be summarized as follows:

Example 10. A model begins with a keyword `machine` followed by a string literal, an opening bracket, and a closing bracket. State definitions are placed between the brackets. A state definition is either a declaration of initial state, or a proper state definition. A declaration of initial state consists of the keyword `initial` followed by an identifier. A proper state definition consists of a keyword `state` followed by an identifier and a pair of square brackets. Transition definitions are placed between the brackets. Each transition definition starts with keywords `on input` followed by a string literal, a keyword `output`, followed by a string literal, followed by keywords `and go to`, followed by an identifier (a target state name).

Ouch! This was quite hard to read! We definitely need a better way to express specifications like the above. Writing them in English seems very cumbersome. We normally do not. Still, being able to describe syntax in natural language is a useful skill. It shows that you can conceptualize syntax, it makes formalization easier, and helps to explain the language to fellow developers. Thus if you have no prior experience with defining syntax, it makes sense to try this on several examples.

Exercise 4.2. Following the style of the `fsm` example above, describe the lexical and syntactic structure of a part of the Google Protocol Buffers language. Use the fragment of the language visible in Fig. 1.4 on p. 10.

The software language engineering community agrees that concrete syntax should be specified using *regular expressions* (for the lexical structure) and formal *grammars* (for the syntactic structure). Regular expressions seem to capture most of the necessary constraints for tokens in software languages, and grammars do the same for syntactic structure. We briefly recall the essence of both formalisms below.

Regular expressions. You are probably familiar with regular expressions from scripting languages, web programming practice, or advanced search facilities in development editors. Regular expressions found in real languages and tools tend to be complex and rich. The good news is that a tiny fully expressive core hides inside that has it all. It turns out that there are only three operators in the core language! It is useful to appreciate this minimal core language to internalize the limitations and use cases of regular expressions.

Regular expressions are defined given a fixed finite set of characters, an *alphabet*. In modern practice, the alphabet is typically a variant of Unicode. However, since regular expressions can be used to describe other things than program text, let us assume that we use a finite set Σ of unknown symbols.

Example 11. Binary numbers are numbers that are written using only two digits: zero (0) and one (1). We will write a regular expression defining what is the syntax of a binary number. For this example, we take the alphabet to be all letters and digits:

$$\Sigma = \{0,\dots,9,a,\dots,z\} \ ,$$

although we will only use the first two digits in our expression. Other characters might be used in describing other tokens of our language.

Do not get discouraged by the abstract nature of this example. We chose binary numbers mostly because zeroes and ones are easy to write on paper. Nevertheless, binary patterns have many applications. Imagine instead that we are designing a language where we want to describe Morse code messages. A zero may represent a short tone and one may represent a long tone. A token in our language can represent a coded letter. Or consider a computer game, where users are allowed to define their own textures to create new tiles. One possible texture definition is via monochromatic patterns. A token of zero-ones could represent an alternation of black and white points in the texture.

The regular expression 0 represents a word consisting just of zero, and the expression 1 represents a word consisting just of the digit one. A single symbol from the alphabet Σ is a regular expression and it means a string that contain exactly one symbol, this digit. Below, we use the double square brackets to denote the meaning of an expression. Note that a meaning of a regular expression is a set of tokens (strings), so the meaning of 0 is a set containing a single-character string with zero:

$$[\![0]\!] = \{\text{"0"}\} \tag{4.3}$$

$$[\![1]\!] = \{\text{"1"}\} \tag{4.4}$$

We write ε to represent an empty (zero-length) word that contains no characters. This might sound weird at first, but an empty word is rather useful. It may for example represent an empty pattern texture (which could mean transparency in our game). It is also useful for keeping the regular expression notation small, as it allows many interesting constructs to be derived (see below). In practical implementations ε is typically written as “nothing” (no character), but in the definition below we will write it out explicitly for clarity.

$$[\![\varepsilon]\!] = \{\text{""}\} \tag{4.5}$$

We can build more complex expressions to describe *longer words* or tokens by *concatenating* simpler expressions *sequentially*. For instance, 00001111 represents the word of four zeros followed by four ones, 01 may represent the letter A in Morse code, and 00000000 can represent a completely black fragment of a texture in our game.

$$[\![00001111]\!] = \{\text{"00001111"}\} \tag{4.6}$$
$$[\![01]\!] = \{\text{"01"}\} \tag{4.7}$$
$$[\![00000000]\!] = \{\text{"00000000"}\} \tag{4.8}$$

Languages using only one possible term are not interesting. In our example, we need to describe not a single token, but *any* binary number. To describe bigger sets of tokens, we can combine simpler expressions with the *alternative* operator denoted by the pipe symbol. For example, $\varepsilon \mid 01 \mid 1000$ means a set of three tokens (in Morse code: no letter, letter A, or B):

$$[\![\varepsilon \mid 01 \mid 1000]\!] = \{\text{""}, \text{"01"}, \text{"1000"}\} \tag{4.9}$$

Even with the alternative operator, we can only describe finite numbers of tokens. Worse, our regular expressions are as large as the languages they describe. We cannot possibly enumerate all binary numbers in a single expression! We need an iteration constructs to define larger sets. This is a task for the *Kleene closure* operator, denoted by a post-fix plus sign. An expression 1^+ means any non-empty sequence of 1s (a unary number). We describe a binary number by combining the Kleene closure with alternative: $(1 \mid 0)^+$. The meaning of the two expressions is:

$$[\![1^+]\!] = \{\text{"1"}, \text{"11"}, \text{"111"}, \ldots\} \tag{4.10}$$
$$[\![(0|1)^+]\!] = \{\text{"0"}, \text{"1"}, \text{"00"}, \text{"01"}, \text{"10"}, \text{"11"}, \ldots\} \tag{4.11}$$

Let us gather the constructs in a formal definition of the notation of regular expressions.

Definition 4.12 (Syntax of Regular Expressions)**.** *Let Σ be a finite alphabet and let ε denote the empty sequence. Then*

Base case (simple expressions):

ε	*is a regular expression*
a	*is a regular expression for any symbol $a \in \Sigma$*

Let r, s be regular expressions. Then (inductive case):

$r \mid s$	*is a regular expression (union)*
rs	*is a regular expression (concatenation)*
r^+	*is a regular expression (Kleene closure)*

A regular expression generates a set of tokens over alphabet Σ according to the following rules.

Definition 4.13 (Semantics of Regular Expressions)**.**
Base case (simple expressions):

$$[\![\varepsilon]\!] = \{\varepsilon\}$$
$$[\![a]\!] = \{a\} \qquad \textit{for any } \; a \in \Sigma$$

Inductive case (composite expressions):

$$
\begin{aligned}
[\![r \mid s]\!] &= [\![r]\!] \cup [\![s]\!] && \textit{(alternative or union)} \\
[\![rs]\!] &= \{ vw \mid v \in [\![r]\!] \wedge w \in [\![s]\!] \} && \textit{(concatenation)} \\
[\![r^+]\!] &= \{ v_1 \ldots v_n \mid v_i \in [\![r]\!], 1 \leq i \leq n, n \in \mathbb{N} \} && \textit{(Kleene closure)}
\end{aligned}
$$

The first inductive case in Def. 4.13 says that the generated language contains any word generated either by r or by s. The second case means that the generated language contains languages created by concatenating any word from the language generated by r with any word from the language generated by s. The last case, the Kleene closure, should be read as follows: the generated set contains words created by concatenating any positive number of words from the language generated by r.

It turns out that the above definition is complete—it defines regular expressions able to generate any regular language, so any language recognized by a finite automaton. If you are interested in learning more about its theoretical properties, please refer to a more foundational text on the theory of automata (for example Hopcroft, Motwani, and Ullman [13]). For us, this simply means that we can define essentially any relevant token using the above constructs. Try the following exercise.

Exercise 4.3. Write a regular expression defining binary numbers without (left) leading zeroes. The only binary number with leftmost zero allowed is zero itself. Only use the regular expression operators introduced above. *Positive examples:* 0, 10, 11, 10101011; *Negative examples:* 00, 01, 011, 0000000

Table 4.1 lists several extensions[1] to regular languages known from scripting languages and other operating systems tools. Moreover, it shows that these are, in fact, *syntactic sugar* of our simple core subset; they allow us to write more conveniently things already possible in the language of Def. 4.12. Syntactic sugar is of course important for users, and when you are defining the lexical structure of languages, you definitely want to use such extensions. Similarly, you want to add convenience syntax to your own language; thus we will return to adding syntactic sugar to your language below.

Context-free grammars. We shall use *context-free grammars* (CFGs) to describe syntax of correct models in a DSL. A specification of concrete syntax should facilitate translation from textual input to abstract-syntax trees. Yes, the core structure of any abstract-syntax representation is a tree

[1] See for instance https://www.regular-expressions.info/posixbrackets.html, accessed 2022/09

Operator name	Expression	Expansion
optional	$r?$	$r \vert \varepsilon$
Kleene star	r^*	$r^+ \vert \varepsilon$
character range	$[a-zA-Z]$	$a \mid \cdots \mid z \mid A \mid \cdots \mid Z$
alphanumeric symbol	[: alnum :]	$[a\text{–}zA\text{–}Z0\text{–}9]$

Table 4.1: Examples of syntactic sugar extensions of regular expressions

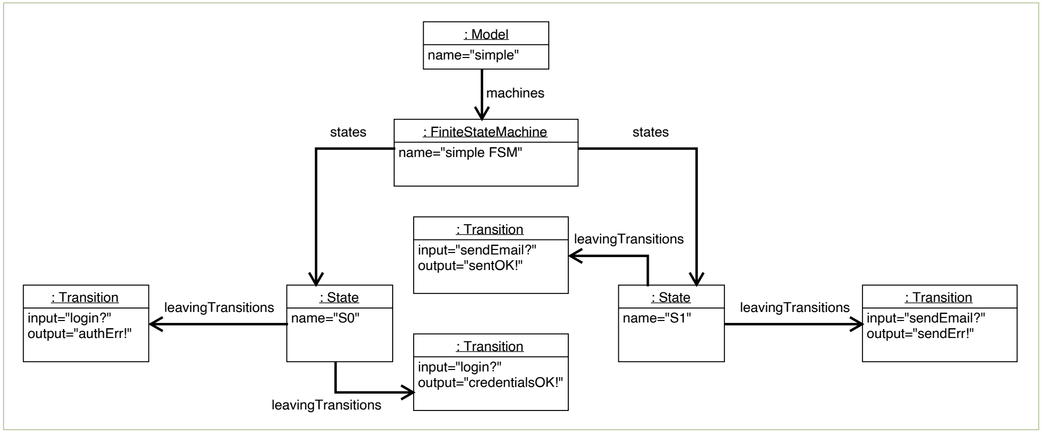


Figure 4.2: *A decomposition of the abstract-syntax instance shown in the top of Fig. 4.1. Convince yourself that it is an instance of the partonomy of Fig. 3.4*

(cf. Fig. 4.2). Intuitively, since we need to create trees, we need a formalism that will be able to "see" the input as trees.

There is a tree in disguise in most structured computer text (models and programs). The most obvious tree structure is given by nesting of parentheses. For example, in Fig. 4.1 (bottom right), square brackets show that transitions are nested in states and that states are nested in machines. Another kind of nesting is defined by property–object relationships. In the figure, a machine *has* a name, and a transition *has* an input, an output, and a target state; so properties are *nested* under the larger objects.

How can we describe trees hiding in the textual input? We do this inductively! We define what are the leaves (the base case) and what are the inner nodes—for each node type we say what are the possible children (the inductive case). Grammars are exactly the formalism that allows us to describe such an inductive generation of trees. Consider the example below.

Example 12. Let us develop intuition for how grammars capture program text by analyzing syntax of arithmetic expressions, a small language with a rather natural inductive structure. Assume that expressions can be written with use of variable names, and two operators: multiplication ($*$) and addition ($+$), for example $x+y*z$. This expression is captured by the following expression tree. You have probably seen similar expression trees in primary school, not realizing that they were abstract-syntax trees:

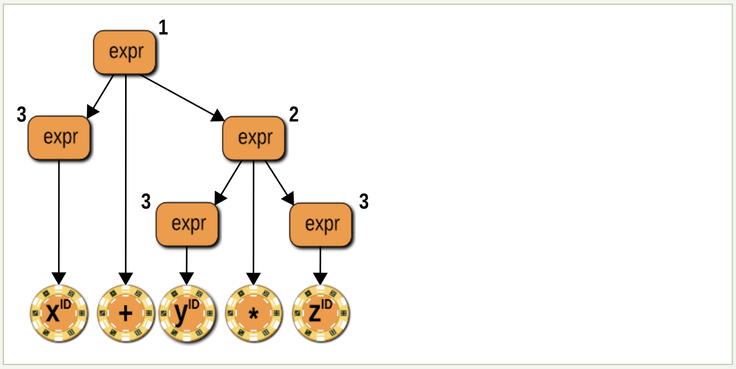


Figure 4.3: An abstract-syntax tree for the expression $x + y * z$, an informal notation. Numeric labels indicate a possible derivation order, explained below

In the figure, the leaves are drawn as poker tokens to emphasize that the basic elements in our grammatical statements are lexical tokens, defined by regular expressions. Observe that the tokens ordered from left to right form the original expression $x + y * z$.

How do we specify what arrows we should draw to form the tree on top of the tokens? First, look at the unary nodes (nodes with only one outgoing arrow). In this example, they all happen to be basic nodes, pointing to leaves. We can capture this in a grammar by saying that an expression can be an identifier:[2]

$$\text{expr} \rightarrow_3 \texttt{ID} \ , \tag{4.14}$$

where expr stands for a piece of text that is an expression, and `ID` means a token representing a variable name (an identifier).

What about the two remaining ternary nodes? For them we have to specify the branching: what three components are allowed to be nested under them. It turns out that we have two kinds of them, one for addition, and one for multiplication. Each allows first a left sub-expression, then an operator token, and a right sub-expression:

$$\text{expr} \rightarrow_2 \ \text{expr} \ \ \texttt{'*'} \ \ \text{expr} \tag{4.15}$$

$$\text{expr} \rightarrow_1 \ \text{expr} \ \ \texttt{'+'} \ \ \text{expr} \tag{4.16}$$

Note how this structure with nesting sub-expressions (instead of identifiers directly) allows us to represent larger and larger expressions inductively. For example, the same rules can be used to generate a sum of sums of multiplications.

The above three rules allow us to generate arbitrary expression trees in the language of arithmetic expressions with addition and multiplication. The keyword are to *generate* or to *derive*, as we apply the rule from left to right, creating longer and longer strings. Here is an example of a derivation, with labels on arrows denoting which rule has been applied (they also correspond to the labels in Fig. 4.3):

$$
\begin{aligned}
\mathsf{expr} &\rightarrow_1 \texttt{expr '+' expr}\\
&\rightarrow_2 \texttt{expr '+' expr '*' expr}\\
&\rightarrow_3 \texttt{expr '+' expr '*' ID}\\
&\rightarrow_3 \texttt{expr '+' ID '*' ID}\\
&\rightarrow_3 \texttt{ID '+' ID '*' ID}
\end{aligned}
\tag{4.17}
$$

In the above, we always expand the rightmost occurrence of expr using one of our rules, as labeled on the arrow. If you start drawing the tree in the same order, you will obtain the same image as in Fig. 4.3. Such a sequence of expansion steps is called the rightmost derivation of the string of tokens.

We define context-free grammars for a fixed finite set of symbols denoted T (for 'tokens'). In grammars, the basic symbols are entire tokens, unlike in lexical specifications where they tend to be characters.

Definition 4.18 (Syntax of Context-Free Grammars)**.** *Let T be a finite set of* terminal symbols *(tokens), and let N be a finite set of* non-terminal symbols *(syntactic categories). A* grammar production rule*, or a* production *for short, is a pair of a non-terminal symbol $n \in N$ and a sequence σ of terminal and non-terminal symbols $\sigma \in (N \cup T)^*$. We typically write a production (n, σ) using an arrow, emphasizing that the sequence σ can be* derived *or* generated *from the non-terminal n: $n \rightarrow \sigma$.*

A context-free grammar *(CFG) over sets of terminal (T) and non-terminal (N) symbols is a set of production rules over T and N, with a dedicated start non-terminal $s \in N$.*

In other words, to write a grammar, we need to choose a set of tokens and specify left-to-right productions generating strings of these tokens. The meaning of the productions, so which language they define, is explained in the semantics of context-free grammars.

Definition 4.19 (Semantics of Context-Free Grammars)**.** *Assume that s is a start non-terminal of a context-free grammar G, with a production relation $\rightarrow \subseteq N \times (T \cup N)^*$. Then the grammar G* generates *a language of words (sequences) over the alphabet of terminals T as follows:*

$$[\![G]\!] = \{w \in T^* \mid s \rightarrow^* w\} \tag{4.20}$$

where $\rightarrow^*$ denotes the reflexive transitive closure of relation $\rightarrow$.

In simple words, the language $[\![G]\!]$ defined by the grammar G contains all the models that can be created by expanding the start symbol by repeated application of production rules, until all non-terminal symbols are eliminated.

Why do we call these grammars "context-free"? Recall the format of the grammar rules: a production is applied to any non-terminal symbol in a

[2]For the time being, ignore that we need to distinguish precisely what identifier we are seeing. We will come back to this in Sect. 4.4

The Unusual Past of Formal Grammars

Source: Wikipedia user Σ (CC BY-SA 4.0)

Noam Chomsky (born 1928) is an American linguist, philosopher, and political activist. Chomsky created a formal theory of *transformational generative grammars* to understand natural languages. As a linguist, Chomsky was not particularly interested in programming and modeling languages—he studied the structure of natural languages used by people to communicate. Chomsky defined a hierarchy of increasingly expressive ways to specify languages using grammars, known today as *Chomsky's Hierarchy*. The least expressive languages in the hierarchy are the regular languages (generated by familiar regular expressions). Context-free languages (generated by context-free grammars) take the second level, followed by context-sensitive languages, and recursively enumerable languages. This work was published in a highly influential volume called *Syntactic Structures* [9].

Today, Chomsky's work remains one of the foundations of theoretical computer science, while the specification formalisms he introduced are the staple of software language engineering work.

sequence, without taking into consideration its context. In Eq. (4.17), we expanded rules 1–3 without considering what precedes and what follows the non-terminal expr. We always choose just one terminal at a time, and expand it by substituting the right-hand side of the production. There exist more complex, context-sensitive, grammars where the productions are applied by considering in what surrounding the expanded non-terminal is placed. These grammars are rarely used in language engineering.

Exercise 4.4. Consider again the grammar for arithmetic expressions used above:

$$\text{expr} \rightarrow_2 \text{expr}\ \texttt{'*'}\ \text{expr} \qquad \text{expr} \rightarrow_1 \text{expr}\ \texttt{'+'}\ \text{expr} \qquad \text{expr} \rightarrow_3 \texttt{ID}\ .$$

Add terminals for the opening and closing parentheses `'('` and `')'`. Extend the grammar to handle parenthesized expressions. How many rules do you need to add?

Exercise 4.5. Consider a simple context-free grammar (small letters denote non-terminals, quoted letters terminals, ε the empty string, and n is the start symbol):

$$\text{n} \rightarrow_1 \texttt{'a'}\ \texttt{'c'}\ \text{b}\ \text{b} \quad \text{b} \rightarrow_2 \texttt{'x'}\ \text{b}\ \texttt{'x'} \quad \text{n} \rightarrow_3 \text{b}\ \text{b}\ \texttt{'a'}\ \texttt{'c'} \quad \text{b} \rightarrow_4 \varepsilon\ .$$

Does the word `'acxxxac'` belong to the language generated by this grammar? Argue why not, or show a derivation of the string from the start symbol.

So far, we have strictly separated the use of grammars and regular expressions. We used regular expressions to define tokens (the lexical structure) and the grammars to define the overall syntax. However, both in practice and in theory these two notations overlap significantly. As noted in Chomsky's Hierarchy (info box on p. 97), every regular language is a context-free language. This means that we can rewrite every regular expression over an alphabet Σ to a context-free grammar with Σ being the set of terminal symbols. Can you?

Exercise 4.6. Translate the regular expression from Exercise 4.3 (p. 93) to a context-free grammar. If you skipped that exercise, simply write from scratch a CFG generating the language of binary numbers without leading zeroes. The terminal symbols should be `'0'` and '1'. **Hint:** The translation rules from regular expressions to CFGs are listed below.

The translation rules for core regular expressions to context-free grammars are quite simple. The expressions in extended regex languages can be reduced to context-free grammars by first expanding their syntactic sugar (Tbl. 4.1) and then applying the rules below.

1. A regular expression r generating a single alphabet symbol, say `'r'`, is translated to a production with a single token representing the same symbol. We need to invent a non-terminal symbol to be placed on the left-hand side, say: R' $\rightarrow$ `'r'`.
2. A regular expression $r|s$ is translated to two productions: RS' $\rightarrow$ R and RS $\rightarrow$ S, where RS' is a fresh nonterminal, and R, S are the nonterminals created during translation of r and s (inductively).
3. A regular expression rs is translated to a single grammar production: RS' $\rightarrow$ R S, where RS' is a fresh nonterminal, and R, S are the nonterminals created during translation of r and s (inductively).
4. A regular expression r^+ is translated to two productions: R' $\rightarrow$ R R' and R' $\rightarrow$ R, where R' is a fresh non-terminal, and R is the nonterminal created during translation of r (inductively).

Why do we bother to learn and use regular expressions, if all the same could have been achieved with grammars? Primarily, because the regular expression notation is so concise and convenient. Even when you are writing grammars, it is convenient to use some regular expression operations. For example, it is much easier to write a regular expression for a list of objects with the same syntax (just use Kleene star) than to devise the appropriate productions. You need two grammar productions to express this simple case. Try!

For this reason, researchers have defined an extended notation for CFGs, the Extended Backus-Naur Form (EBNF for short). EBNF includes the regular expression operators as syntactic sugar. The essence of the EBNF notation is summarized in Tbl. 4.2. EBNF became very popular. It is the basis of the specification language of most modern syntax design tools.

Exercise 4.7. Write a simple EBNF grammar for an expression language with variables, and conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). Assume that terminals `Id`, `Not`, `And`, and `Or` are defined. They match, in the following order: variable identifiers, negation, conjunction, and disjunction operators. Your grammar should be able to generate, among others, the following example expression: $x \wedge \neg(y \vee (z \wedge x \vee \neg y))$.

4.3 How to Write a Grammar in Practice

Having seen the basic specification notations, we want to understand how specifications are created in practice. We shall observe the process on a small case study; beginning with sketches (mock-ups), requirements, and moving to identification of tokens, nonterminals, and rules. Even though we want to be practical, we still use only classic EBNF and regular expressions. We shall move to real tools in Sect. 4.4. Especially for new languages, it is useful to lay down the initial construction of concrete syntax sidestepping the accidental complexity brought by tools. A *base-line grammar* is best created using fundamental notations [16], especially in learning situations.

Develop mock-ups. In practice, designing and specifying concrete syntax is not as formal as it would seem based on the above definitions. Expressing the design for a language in formal notation is cumbersome, especially if the usability is to be assessed. It is easier and more effective to create mock-up models in the envisioned syntax. Mock-up models may be shown to stakeholders and discussed. Before you start writing grammars and regular expressions, ask yourself what the models in your language will look like. Revisit the last question from domain analysis: *Do we have any examples of existing notation? Use the existing examples and create new* (cf. Tbl. 3.1).

Example 13. For the finite-state-machine language (`fsm`), we may want to base a textual syntax on the familiar mathematical definition of a state machine: *a state machine is a triple—a set of states, an initial state, and a transition relation.* The left part of Fig. 4.4 shows how such a syntax could look. We start with defining and naming a finite set of states (Line 1), proceed to define a transition relation (l. 3–8), and use these elements to declare a state machine (l. 10).

This notation is very concise. The key control structure of our state machine is covered in just four lines (4–7)! On the other hand, this syntax is not very scalable. If we had many states, the flat list of transitions would not resemble an automaton at all. Our target audience (CS students) might find this notation alien, too remote from programming languages.

We recommend to create several prototypes of the syntax, and possibly several variations of the most interesting prototypes, before you start any implementation. Prototyping is very easy (use paper, or a generic text editor) and it provides instantaneous and valuable feedback. Consider another design for the `fsm` language.

EBNF operator	EBNF production	CFG productions
alternative	$S \to \alpha \mid \beta$	$S \to \alpha$ $S \to \beta$
optional	$S \to \alpha\ T?\ \beta$	$S \to \alpha\ T'\beta$ $T' \to T \mid \varepsilon$
iteration	$S \to \alpha\ T^{+}\ \beta$	$S \to \alpha\ T'\beta$ $T' \to T\ T'?$
grouping	$S \to \alpha\ (\beta)\ \gamma$	$S \to \alpha\ T'\ \gamma$ $T' \to \beta$

Table 4.2: Extended Backus-Naur Form (EBNF) for context-free grammars, defined as a syntactic sugar of Chomsky's CFGs. α and β stand for arbitrary sequences of terminals and nonterminals. Kleene star can be expanded to iteration just like for regular expressions

```
let State = { S0, S1 }

let Tran = {
  transition (S0, "login"?, "credentialsOK"!, S1)
  transition (S0, "login"?, no!, S0)
  transition (S1, "sendEmail"?, "sendErr"!, S1)
  transition (S1, "sendEmail"?, "sendOK"!, S0)
}

let simpleFSM = machine (State, S0, Tran)
```

```
machine "simple FSM" [
  initial S0
  state S0 [
    on input "login" output "credentialsOK" and go to S1
    on input "login" output "authErr" and go to S0
  ]
  state S1 [
    on input "sendEmail" output "sendErr" and go to S1
    on input "sendEmail" output "sendOK" and go to S0
  ]
]
```

Figure 4.4: *Two sketches of concrete syntax for the language of finite-state machines: mimicking a mathematical definition (left), and programming language style with nested blocks (right)*

Example 14. The right hand side of Fig. 4.4 shows an example of the same `fsm` model in another prototype syntax, where states are used to group transitions. A transition is always nested in its source state, and it is presented in text resembling English sentences. This syntax clearly takes more space, but it does have some advantages. The behavior of each state is always gathered in one place and the blocks enclosed in brackets will appear familiar to programmers. Even though in our language (Tbl. 3.1) states cannot be nested, this syntax would support nested states as a conservative extension, if we needed it one day. Finally, recall that our main use case was to support interpreters. A syntax devoting a line to each state will make it easier to design an animation tool that highlights the active state during the execution.

Obviously, whether the first or the second design is preferred depends on many criteria and on a particular usage context (that is arguably underspecified in our example). Somewhat arbitrarily, we decide to continue with the second variant in the remaining part of this chapter. Exercise 4.46 continues further development of the first, more mathematical, design.

Extend your mock-up against requirements. When creating syntax mock-ups it is useful to inspect corner cases in the language: How are we going to express all the syntactic variations? We can identify and address such questions by systematically scanning the meta-model for variability of properties, or by going through the initial language requirements. For the `fsm` example, we extract the requirements and issues from the domain analysis in Tbl. 3.1 and from the meta-models in Fig. 3.1 and Fig. 3.5. Table 4.3 shows the results of this analysis.

An analysis, like the one in Tbl. 4.3, provides a good opportunity to create a model encompassing all syntactic variations. Always create a possibly complete, large mock-up model and save it for the purpose of testing the parser. Figure 4.5 shows such a model for the `fsm` case. Notice that this representation of a finite-state machine is actually a Scala program. The model is stored in a value (`m`) and it does not use square brackets, which in Scala are reserved for type annotations. With a suitable library implementing this DSL we can actually compile, and eventually execute, this model just using a Scala compiler (including the parser) and runtime environment.

Requirement	Justification if met, extension otherwise	Example syntax
Can we represent initial states?	A declaration of an initial state is shown in l. 3 (Fig. 4.4, right).	`initial`

Can we represent end states?	The meta-model allows states without outgoing transitions. This can be written in the mock-up syntax using empty brackets. Allowing an empty list of transitions is a new requirement though. It would be good to allow omitting the brackets.	`state S2 []`
		`state S3`

States and machines should have names.	Both states and machines are named in Fig. 4.4 (lines 1, 5, 10), but some names are quoted and some are not. Uniformize the design and allow quoting state names (plus spaces in names).	`state "state S4" []`

Transitions should have inputs.	The mock-up transitions already have inputs, but all inputs are quoted. Relax this requirement to uniformize with state names. When input names are not quoted, it seems natural to drop the `input` keyword, too. Let's make it optional.	`on input sendEmail`
		`  output sendErr and go to S1`
		`on sendEmail`
		`   output sendOK and go to S0`

Can we represent optional outputs?	Our mock-up example always includes an output label. Let us add another example of a transition to show syntax without output labels. When omitting an output, we would like to skip the `and` keyword to avoid awkwardness, as shown to the right.	`on shutDown and go to S3`
		`on shutDown go to S3`

Table 4.3: *The mock-up `fsm` syntax of Fig. 4.4 (right), against the requirements of domain analysis*

Once mock-ups are created we begin to design the grammar. Let us start with tokens.

Identify tokens. We enumerate all token kinds used in the mock-up example (Fig. 4.5), grouped by their role in the syntax in Tbl. 4.4. The three categories of tokens in the table are typical for most languages. Punctuation can be further divided into separators (comma, colon, semicolon), operators (plus, minus, navigation dot), and delimiters (parentheses, brackets, braces, quotes). Besides these, one typically would like to allow *comments* (often handled as white space in the definition of lexical structure) and *literals* (string literals, integer and floating-point number literals, etc.) We keep the list of tokens very small for this example language for brevity.

> **Exercise 4.8.** Write a regular expression defining the tokens of signed integer literals. A literal constant cannot start with a zero, except for the constant 0 itself. The sign is optional. *Positive examples:* +1, 231, -0, -999999999. *Negative examples:* 001, 0009, -099999, +01

Specify terminals. We begin specifying the syntactic structure of a language by defining its terminal symbols. For every fixed token (punctuation and keywords) we create a terminal symbol representing it. With most tools—we can write regular expressions defining the tokens from the previous step directly in the grammar. Table 4.4 lists the terminal symbols of our grammar in the rightmost column.

Category	Tokens	Regular expression	Terminal symbol
Keywords	`machine`	`'machine'`	Machine
	`initial`	`'initial'`	Initial
	`state`	`'state'`	State
	`on`	`'on'`	On
	`input`	`'input'`	Input
	`output`	`'output'`	Output
	`and`	`'and'`	And
	`go`	`'go'`	Go
	`to`	`'to'`	To
Punctuation	`[`	`'['`	LBracket
	`]`	`']'`	RBracket
Identifiers	`"complete FSM" "login" "credentialsOK" "authErr" "sendEmail" "sendErr" "sendOK" "state S5" S0 S1 S2 S3 S4 S5.`	`("[a-zA-Z]([:alnum:]|' ')*") | ([a-zA-Z][:alnum:]*)`	Id

Table 4.4: *Token categories in the* `fsm` *case study.*

```
machine "Complete FSM" [
  initial S0
  state S0 [
    on input "login"  output "credentialsOK"  and go to S1
    on input "login"  output "authErr"        and go to S0
  ]
  state S1 [
    on input "sendEmail"  output  "sendErr"  and go to S1
    on input "sendEmail"  output  "sendOK"   and go to S0
  ]
  state S2 []
  state S3
  state "state S4" [
    on input sendEmail output sendErr and go to S1
    on sendEmail output sendOK and go to S0
  ]
  state S5 [
    on shutDown go to S3
    on input shutDown go to S3
  ]
]
```

source: fsm/test-files/Complete.fsm

Figure 4.5: *A larger* `fsm` *syntax mock-up, created as a test case for a parser. This mock-up contains most of the possible variations in syntactic structure*

Handling identifiers and literals is slightly more complex than keywords and punctuation. A keyword carries no interesting information besides that it appears in the program text. An identifier or a literal belongs to a larger category defined by a single regular expression, and we need to remember what is the actual name or constant written by the programmer. For this reason, in tools, a token carries the string value of the matched expression, which can later be mapped to the right type. For instance, a matched identifier or a string literal may be stripped of surrounding quotes, while an integer constant may be converted into an integer value.

Syntactic category	Example	Intuition / Justification
machineBlock	`machine "complete FSM" [ ... ]`	The `machine` keyword (Fig. 4.5) initiates a block encompassing the entire file that describes a machine. It clearly corresponds to the meta-model concept FiniteStateMachine in Fig. 3.1.
initialDeclaration	`initial S0`	Line 2 declares that state S0 is initial. It seems logical to make this declaration separate from the definition of state S0 in the following lines. Eventually, it should correspond to the initial reference in Fig. 3.1. The state has an optional block of transitions in the last part. We probably need a transition concept, too.
stateBlock	`state S2 [ ... ]`	The example contains six state definition blocks (`S0`–`S5`). They all seem to have a similar structure, and correspond to the State concept in the meta-model.

transition	`on input "login"`
	`  output "authErr"`
	`  and go to S0`
		Transitions come in a number of variations, but there is no doubt that they are all an instance of the same concept, the Transition meta-class in the meta-model. It appears that each transition line has up to three parts: input, output, and a target state. Since some of these are optional, it is useful to think about them as separate syntactic elements (below).

inputClause	`on input "login"`	Specifies the input to which the transition reacts. It will populate the input property of the Transition meta-class.
outputClause	`output "authErr" and`	Specifies the output that the transition produces; will populate the output property of the Transition class.
targetClause	`go to S0`	Specifies the target state of a transition. It will be used to set the target reference in the meta-model.

Table 4.5: Syntactic categories (larger than one token) extracted from our examples, cf. Figures 3.1 and 4.5

A careful reader has noticed that there seems to be a conflict between the definitions of keywords and identifiers in the `fsm` example. (Can you spot it in Tbl. 4.4?) All our keywords are also identifiers! For instance, "`machine`" is technically also a legal state name. Naming a state a `machine` could be extremely confusing! Fortunately, this is not a problem in practice. Most parsers allow tokens to be prioritized, so that keywords should be matched first, and pre-empt any possible matching of an identifier, if a keyword match succeeds. So any identifier specification in a language definition has an implicit condition that the matched string does not match any other token with a higher priority. Simple tokens, such as keywords and punctuation, tend to be assigned the highest priority.

Exercise 4.9. Identify tokens (and categories of tokens) in the example model in the `robot` language shown in Fig. 2.2 (p. 30). Formulate regular expressions for the identified categories of tokens. The exercise should result in a table similar to Tbl. 4.4, but for the `robot` language.

Identify syntactic categories. We shall now define the syntactic structure of our language. This is more difficult than understanding what tokens are used. We need to infer the structure from the examples. We shall proceed by listing the *syntactic categories*, so groups of adjacent tokens that represent a single concept in the model. Usually the nesting structure allows us to discover some of these, while others appear because they are logically cohesive. Table 4.5 lists syntactic categories that are easy to spot in the example of Fig. 4.5.

Inferring syntactic categories from examples is a rather difficult process that requires intuition and experience. Typically, only key syntactic categories are easily visible, and nonterminal symbols can be defined for them (below). Normally, you will discover the missing categories when specifying the grammar. Of course, in practice we never write out the syntactic categories with the level of detail of Tbl. 4.5. An experienced grammar writer makes such observations on-the-fly, while writing the grammar productions. On the other hand, if you are new to the graft of grammar specification, this might be a useful exercise.

Specify grammar rules. Once you know the syntactic categories of your language, writing the grammar productions is quite easy. Syntactic categories become nonterminals, tokens become terminals, and your syntax specification governs the rules (which should cover the examples you generated by now). We begin with several simple rules from the bottom of Tbl. 4.5:

$$\begin{aligned}
\mathsf{inputClause} &\rightarrow \texttt{'on'}\ \texttt{'input'}?\ \texttt{Id}\\
\mathsf{outputClause} &\rightarrow \texttt{'output'}\ \texttt{Id}\ \texttt{'and'}\\
\mathsf{targetClause} &\rightarrow \texttt{'go'}\ \texttt{'to'}\ \texttt{Id}\\
\mathsf{transition} &\rightarrow \mathsf{inputClause}\ \ \mathsf{outputClause}?\ \ \mathsf{targetClause}\\
\mathsf{stateBlock} &\rightarrow \texttt{'state'}\ \texttt{Id}\ (\ \texttt{'['}\ \ \mathsf{transition}^*\ \ \texttt{']'})?\\
\mathsf{initialDeclaration} &\rightarrow \texttt{initial Id}
\end{aligned} \tag{4.21}$$

Exercise 4.10. Explain how the Kleene star ($*$) and the optional operator above (?) interact to provide two possible syntactic ways to specify an end state.

The above rules were rather simple to specify, but now we reach a stumbling block: the initial state declaration (`initial S0` below) should be allowed to be placed anywhere within the machine block. At the same time, we would like to make sure that at least one state and exactly one initial state are specified. We could try the following sequence of grammar symbols:

$$\mathsf{stateBlock}^*\ \ \mathsf{initialDeclaration}\ \ \mathsf{stateBlock}^* \tag{4.22}$$

This enforces that exactly one initial declaration is placed within a sequence, while *some* state blocks are *allowed* before and after. It does not *guarantee* though that at least one state is defined (one state block is included). This single state could be defined either before or after the initial declaration, so we need to change the Kleene star operation on one of them to a Kleene plus. But which one? If we want to be entirely flexible, then we should allow both options: the state block must appear either in front of or after the initial declaration. Now this piece of grammar becomes large enough to give it a non-terminal name:

$$\begin{aligned}\mathsf{machineBlockContents} \;&\rightarrow \big(\;\mathsf{stateBlock}^{+}\;\; \mathsf{initialDeclaration}\;\; \mathsf{stateBlock}^{*}\big)\\ &\quad\mid \big(\;\mathsf{stateBlock}^{*}\;\; \mathsf{initialDeclaration}\;\; \mathsf{stateBlock}^{+}\big) \qquad (4.23)\end{aligned}$$

Defining the shape of allowed inputs precisely quickly becomes quite cumbersome. It is typically better to settle for simple, approximating presentations like Eq. (4.22). This has several advantages. Smaller grammars are easier to maintain and debug. Also better error messages can be produced if detection of detailed misformulation is done later, in the static analysis phase using a type checker or constraints (see Chapter 5).

Finally, we use the new non-terminal to specify the machine blocks. We also add a new non-terminal, the start symbol, defining the entire model with multiple machines:

$$\begin{aligned}\mathsf{machineBlock} \;&\rightarrow\; \texttt{'machine'}\;\; \texttt{Id}\;\; \big(\;\texttt{'['}\;\; \mathsf{machineBlockContents}?\;\texttt{']'}\big)?\\ \mathsf{model} \;&\rightarrow\; \mathsf{machineBlock}^{*} \qquad (4.24)\end{aligned}$$

We conclude the section by summarizing the six-step method which we used for writing down the `fsm` grammar:

1. **Develop mock-up examples.** Writing examples is easier than writing grammars. You can experiment faster with examples, and show them to customers before you commit to an implementation.
2. **Extend mock-ups against requirements.** Collect all the requirements you can, from domain analysis, and from interacting with customers. In the end, create a large comprehensive example and use it for testing.
3. **Identify tokens.** Group tokens into categories. These categories are similar across most languages. Parsing tools often offer predefined tokens.
4. **Specify terminals.** Use predefined tokens from your tool, and add the missing regular expressions yourself.
5. **Identify syntactic categories.** Exploit nesting (brackets, parentheses, known structures like expression trees), seek for cohesive concepts, and check that your meta-model concepts are represented in the grammar.
6. **Specify grammar rules.** At this stage, most rules are simple. When some rule is hard to write precisely, consider writing a more permissive rule instead. We can still detect erroneous inputs later using name analysis, type checking, and well-formedness constraints (Chapter 5).

4.4 Parsing and Tools

A grammar definition is made operational by turning it into a *parser*. So far, we wanted you to think about grammars as generators of legal models and programs in the language. A parser does the opposite: it checks (recognizes) whether a given input model belongs to the language; whether it could have

```
grammar dsldesign.fsm.xtext.Fsm
    with org.eclipse.xtext.common.Terminals
import "http://www.dsl.design/dsldesign.fsm"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

model returns Model:
   {Model} machines+=machineBlock*;

machineBlock returns FiniteStateMachine:
   {FiniteStateMachine}
   'machine' name=EString ('['
       ( (states+=stateBlock)+
         & ('initial' initial=[State]) // initialDeclaration
         & (states+=stateBlock)* )?
   ']')?;

stateBlock returns State:
   {State}
   'state' name=EString
   ('[' leavingTransitions+=transition* ']')?;

transition returns Transition:
   'on' 'input'? input=EString         // inputClause
   ('output' output=EString 'and')?    // outputClause
   'go' 'to' target=[State|EString];   // targetClause

EString returns ecore::EString:
   STRING | ID;
```

source: fsm.xtext/src/main/java/dsldesign/fsm/xtext/Fsm.xtext

Figure 4.6: *The grammar for the finite-state-machine language, as described in this chapter, expressed in the input language of the Xtext workbench*

been generated by the grammar. While doing that, it constructs an abstract-syntax tree, or a meta-model instance, representing the input as a data structure in memory. For example, it takes a representation like in the bottom-right corner of Fig. 4.1 and turns it into the instance shown in the top of the figure. In addition, advanced parsers perform *name resolution* and *linking*, turning identifiers of model elements into references to identified objects. For instance, in Fig. 4.1 the parser has turned the token `S0` on line 3 into a reference `initial` from a `FiniteStateMachine` object to a `State` object.

Definition 4.25. *A* parser *is a tool that checks whether an input is syntactically correct and constructs an abstract-syntax representation if so.*

Parsers are rarely written from scratch. Instead we use parser generators and interpreters (combinator libraries). These tools need more information than a plain context-free grammar provides. EBNF just describes how to structure input symbols into trees. A parser needs to know also what types to construct and how to initialize the properties of abstract-syntax objects. An extended notation to specify languages is needed. Unfortunately, there is no agreement on parser specification languages, besides using EBNF as a core. This makes it difficult to present them systematically, in a tool-oblivious manner, in a textbook. As the second-best option, we show two quite different examples below: Xtext (a parser-generator-based language model) and parboiled2 (a parser combinator library for Scala).

In this book, we do not explain in detail how parsers, parser generators, and combinator libraries work (although we do give some pointers to further reading at the end of the chapter). Parsing is a very specialized field of knowledge. Instead, we focus on deriving principles of efficient and practical use of the parsing tools.

An example with Xtext. Xtext[3] is a language workbench, based on the ANTLR[4] parser generator. Xtext is a mature and popular language infrastructure tooling for the JVM platform, popular for both industrial and research-oriented language implementation. Besides parsers, Xtext can generate rich IDE plugins, web editors, language server support[5] [4], and testing infrastructure for your language implementations. Figure 4.6 presents the `fsm` grammar in the syntax of the Xtext input langauge. This grammar specification mixes two kinds of information: how to recognize (generate) a valid model, and how to construct a valid instance of the abstract syntax based on the recognized model.

The first two lines of the example declare the grammar name—the Java package to host the parser code—and import the definitions of standard terminal symbols (typical tokens). Xtext allows grammars to be modularized and reused. In particular, reusing the specification of terminals is very useful as most modern languages share the vast majority of terminals (string literals, numeric literals, identifiers, and operators). Lines 3–4 import the meta-models defining the abstract syntax. We import the `fsm` meta-model (Fig. 3.1) and Ecore. These imports will allow the types of abstract syntax in the grammar productions to be used to construct abstract-syntax objects.

Lines 6–7 define the start symbol, corresponding to the last rule in Eq. (4.24). We used the same identifiers for symbols as in the original production (model, machineBlock), so that the Xtext syntax is directly traceable to our abstract grammar. Colon replaces the right arrow of EBNF. The `returns` clause declares the type of abstract-syntax objects constructed and propagated upwards by the rule. We shall return an object of type `Model`, more precisely an `dsldesign.fsm.Model`. In Line 7, the identifier in curly braces is a *semantic action*, denoting the actual type which will be constructed; this will often be a sub-type of the type specified in the `returns` clause. Xtext will translate this action into a call of the right factory method from the Ecore framework. In this example, the same type is constructed and returned by the rule. In general, when generalization and type hierarchies are used in abstract-syntax definitions these two types may differ. For instance, we may construct a binary expression object, but return upwards an up-cast to an abstract expression type.

Further in Line 7, we match zero or more machine blocks using a Kleene star (`machineBlock*`). Each of the matches will produce an object of type `FiniteStateMachine`—consult lines 9–15 to confirm this. All the

[3] https://www.eclipse.org/Xtext, accessed 2022/09
[4] https://www.antlr.org, accessed 2022/09
[5] https://langserver.org, accessed 2022/09. See also Sect. 2.4

constructed objects will be added to the `machines` collection of the returned `Model` object. Check Fig. 3.1 on p. 53 to convince yourself that a `Model` object indeed has a property `machines`, and that this property is indeed a collection (it can have multiplicity higher than one). The addition of the new object to the `machines` property is another *semantic action* admitted in the Xtext input language.

Definition 4.26. Semantic actions *are executable instructions how to build the abstract-syntax tree. They typically include object constructors, formatting and conversion of the input data into AST format, initialization and updates to properties of the constructed AST, or scoping and name resolution directives.*

Lines 9–15 define a machine block. They correspond to productions in Eqs. (4.23) and (4.24). The grammar elements and semantic actions used are largely the same as in the `model` rule discussed above. We note that the machine identifier is matched using a nonterminal `EString` (see lines 27–28). This allows both quoted and not quoted identifiers as per our requirements. The identifier is stored in the `name` property of the constructed `FiniteStateMachine` object. For convenience, the production defining the machineBlockContents (4.23) has been inlined into the machineBlock rule. This is easier to do in Xtext if a rule does not construct a new object, but merely populates the properties of an already constructed object. More interestingly, it uses the `&` operator of Xtext to specify more succinctly the requirement that the initial state declaration statement has to appear in the machine block, and some states should be defined either before it or after, or on both sides. Compare lines 12–14 to Eq. (4.23). The `&` operator is an unordered composition operator. It admits any sequencing of its operands, which yields a simpler formulation than our original EBNF.

> **Exercise 4.11.** Show that the unordered composition operator `&` of Xtext does not add expressiveness to EBNF, that is, show how to eliminate the operator as a syntactic sugar. More precisely, explain how a production $\mathsf{T} \to \alpha\ (\beta\ \&\ \gamma)\ \delta$ should be transformed to generate the same language, but only using EBNF operators. In the above, `T` stands for a nonterminal symbol, the Greek letters stand for sub-expressions in EBNF. You may want to use Tbl. 4.2 for inspiration.

In the initial-state declaration fragment (Line 13), we meet a new kind of semantic action: a name resolution rule: `[State]`. This action instructs Xtext to match an `ID` token, and to turn the token's value into a reference to an object of type `State`, which has a property `name` holding the same value as the identifier. Thus a name resolution semantic action resolves name-based references into actual references between JVM objects. Technically, this name resolution is performed by Xtext in the second pass, after the parsing has completed successfully and an unlinked abstract-syntax instance has already been constructed. Still, a single specification is used for both purposes.

The remaining productions in Fig. 4.6 use the constructs of Xtext already explained above. You are encouraged to compare them to our abstract

```scala
def model: Rule1[Pure.Model] =
  rule { machineBlock.* ~ EOI ~> Model }

def machineBlock: Rule1[Pure.FiniteStateMachine] = rule {
  "machine" ~ EString ~ BEGIN ~
    stateBlock.* ~
    initialDeclaration ~
    stateBlock.* ~
  END ~> FiniteStateMachine
}

def initialDeclaration: Rule1[String] =
  rule { "initial" ~ EString }

def stateBlock: Rule1[StateTr] =
  rule {
    "state" ~ EString ~ ( BEGIN ~
      transition.* ~
    END ).? ~> StateTransitions
  }

def transition: Rule1[Pure.Transition] =
  rule {
    inputClause ~ outputClause.? ~ targetClause ~> Transition
  }

def inputClause: Rule1[String] =
  rule { "on" ~ "input".? ~ EString }

def outputClause: Rule1[String] =
  rule { "output" ~ EString ~ "and" }

def targetClause: Rule1[String] =
  rule { "go" ~ "to" ~ EString }

def EString: Rule1[String] = rule { ID | STRING }
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/FsmParser.scala

Figure 4.7: The PEG grammar for the finite-state-machine language, as described in this chapter, expressed in the input language of the parboiled2 parser. Only the core part with non-terminal productions is shown here

grammar. Note that the input, output, and target clause productions have (again) been inlined, into the transition rule. The `EString` rule refers to two terminals (`STRING`, `ID`) previously imported from the standard library of Xtext in lines 1–2. If you are interested in learning more about this tool, we recommend the book of Bettini [6].

An example with Scala and parboiled2. For contrast, consider the same example coded in the language of parboiled2,[6] a popular parsing library for Scala. Parboiled2 is a parser combinator library. This means that, unlike Xtext, it is not an independent language, but an internal DSL, so an API exposing the grammar construction operators inside Scala programs. Parboiled2 is implemented using macros, the main meta-programming facility of Scala. Scala macros are executed at compile time. Parboiled2 uses them to generate an efficient implementation of a parser. This is

[6]https://github.com/sirthias/parboiled2, accessed 2022/09

why parboiled2 is very fast, unlike most parser combinator libraries. We will talk more about internal DSLs in Chapter 10 and mechanisms for meta-programming in Chapter 7.

Most parser combinator libraries, including parboiled2, do not parse context-free languages specified by context-free grammars, but use *Parsing Expression Grammars* (PEGs). In general, PEGs and CFGs define two incomparable classes of languages [11]. This means that there exist languages accepted by PEGs, but not by CFGs and most likely vice versa as well. Fortunately, PEGs are stylistically similar to EBNF: the notation and the design process are essentially the same as for CFGs. Thus we can reuse the grammar example from Sect. 4.3 to demonstrate parboiled2. In contrast to CFGs, PEGs are unambiguous—the parsing algorithm is deterministic and fast—but they do lack some of the theoretical succinctness of CFGs, a problem not really experienced in practice. Combinator-based implementations of PEGs, like parboiled2, allow for natural inclusion of arbitrary code into the grammar specification, so expressiveness is not really a problem. The main difference is perhaps in the attitude: a PEG grammar designer should think more in terms of how the text is parsed (recognized), and not how all the legal models in the language are generated. Some of these issues are explored in the exercises in the end of the chapter.

Figure 4.7 presents a parboiled2 PEG for the finite-state-machine language. Contrast it with Fig. 4.6 (Xtext) and with the abstract grammar for finite-state machines of Sect. 4.3. Here, each production is modeled by a single Scala function. We use the same function names as the names of the nonterminal symbols in the context-free grammar. The first function, `model`, defines the start symbol. The presentation format is specific to parboiled2, but most grammars expressed using combinator libraries will look similar.

The grammar specification language of parboiled2 is more flexible than the one used by Xtext, so we have not inlined any rules. We use separate productions for input, output, and target clauses and for the initial state declaration. These were all inlined in the Xtext example in Fig. 4.6. Note that we also add the `EString` rule, in the bottom, mimicking the Xtext style, to match regular and quoted identifiers using a single non-terminal.

Each production is a nullary function (a function that takes no arguments). We explicitly annotate the return types. Thanks to Scala's type inference, these annotations are not strictly required. We include them for clarity, to show what type is constructed by each production. Visually the return types annotations play a similar role to `returns` clauses in the Xtext definition in Fig. 4.6. We have only one kind of rule in this figure, the `Rule1[ ]`. Such rules return a single value, which is placed on the parser stack. Thus `model` and `transition` return the abstract-syntax object representing respectively the entire model and a single transition.

All productions in the example follow the same format: a sequence of grammar symbols is placed within braces after the `rule` keyword (actually a Scala macro). If the value produced by the rule needs to be adjusted, for

```scala
def STRING: Rule1[String] =
  rule { WS.? ~ '"' ~ capture ((!'"' ~ ANY).*) ~ '"' ~ WS.? }

val IDFirst: CharPredicate = CharPredicate.Alpha ++ "_"
val IDSuffix: CharPredicate = CharPredicate.AlphaNum ++ "_"

def ID: Rule1[String] =
  rule { WS.? ~ capture (IDFirst ~ IDSuffix.*) ~ WS.? }

def BEGIN =
  rule { WS.? ~ '[' ~ WS.? }
def END =
  rule { WS.? ~ ']' ~ WS.? }

implicit def StringWS (s: String): Rule0 =
  rule { WS.? ~ str (s) ~ WS }

def WS: Rule0 =
  rule { anyOf (" \n\t").+ }
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/FsmParser.scala

Figure 4.8: The part of the PEG grammar handling the tokens, so what corresponds to a lexer/tokenizer in a classical CFG parser like Xtext/ANTLR

instance to invoke a constructor of a meta-model type, we suffix the rule with a squiggly arrow (~>) and the name of a function implementing the semantic action. Thus in Line 9, the `FiniteStateMachine` is a function name; it is called as the last part of the matching process for the `machineBlock` to execute the semantic action. We discuss an example of a semantic action implementation below. Other notational conventions of interest include: tilde (~) to sequentially combine symbols (white space in classic EBNF), and the navigation dot to attach the otherwise familiar EBNF operators asterisk, plus (Kleene), and question mark. These operators are actually Scala methods. The pipe symbol represents optionality, but unlike in EBNF, it is left-biased, so once a symbol on the left-hand side is matched, the later alternatives will not be considered. This eliminates the ambiguity (non-determinism) issues in PEGs.

In contrast to the Xtext variant of the example, we chose to use the simplified version of the state-sequencing rule (4.22) in the machine block (lines 5–9). This version admits state definitions before and after the initial state declaration, but does not enforce that any definitions are actually included. This simplified rule is easier to read than the one presented in Sect. 4.3 with two alternative choices, but, obviously, this means that we will have to check whether any states are actually declared in later stages. Typically such checking happens during name resolution or static semantic checks (Chapter 5).

Most combinator-based parsers do not have a separate scanner for establishing the lexical structure. We know already from Sect. 4.2 that grammars are sufficiently expressive to replace regular expressions, which proves that they can be used to define the lexical structure of the language as well. This is typically the approach taken by PEG implementers and by parser combinator libraries, including parboiled2. The commonly followed pattern is to write grammar productions to define the tokens. Technically, in these grammars the only terminal symbols are characters of the input set (say Unicode), and all other symbols, tokens and non-tokens alike, are non-terminals.

Figure 4.8 shows the respective part of the Scala example for the finite-state-machine language. We define terminals for quoted strings (`STRING`), identifiers `ID`, opening and closing brackets (`BEGIN`, `END`), and white space (`WS`). The last one is probably the most surprising—since there is no explicit scanner, which would normally filter the white space out, like in Xtext, we need to explicitly mention in the grammar where white space is allowed and required. This is also why all the token productions mention `WS`. Parboiled interprets string and character literals as parsers matching the literal exactly. Since we would like to admit some white space before and require some white space after keywords we modify the default behavior in lines 15–16. This prevents 'glueing keywords' like in `andgoto` (instead of `and go to`).

This listing also includes a new kind of production (`Rule0`). `Rule0` is a type of rule that does not return any interesting value, but consumes some tokens. This is very common for keyword terminals, for white space, and for comments (we do not allow comments in the example).

Lines 4–5 introduce character predicates, which are a compact way to define productions based on character classes. We use them to state what are the legal first characters in an identifier (a letter or underscore), and what are the legal subsequent characters (adding digits to the mix). Both predicates are used in the identifier rule in lines 7–8.

We encourage the reader to study the figure before attempting to solve the following exercise.

Exercise 4.12. Modify the `STRING` definition in Fig. 4.8 to admit special characters using escaping in string literals. In particular, admit `\n` for newline, `\t` for a tabulator symbol, and `\"` for a quote (note that quotes are presently not admitted inside `fsm` strings).

The easiest way to work on this exercise is to modify source code in the book code repository. You can test whether it worked by adding a test case to fsm.scala/src/test/scala/dsldesign/fsm/scala/FsmParserSpec.scala.

Finally, we consider the construction of the abstract-syntax tree by this parser. Since our parser is a pure functional program we cannot easily construct instances of the AST types in `dsldesign.fsm` (Fig. 3.1). This meta-model admits cycles in instances, and it is not possible to construct cyclic structures of references in a purely functional manner without using laziness. Instead the parser uses the pure variant of the meta-model shown in Fig. 3.5, built with algebraic data types. This meta-model is simpler, and does not actually ensure that there are no dangling references in the instances. For instance, the meta-model construction will not detect that a non-existent state has been selected as an initial state of a machine. To produce an instance of the Ecore meta-model we need to perform an additional transformation and checking. We will discuss such transformations in Chapter 7.

The construction of instances of the meta-model types happens in semantic actions. Consider the rule for transition (lines 22–25 in Fig. 4.7) as an

Parser Combinator Libraries

```
    transvbphrase = !transverb --- jointermphrase          >> apply2
                  | !linkingverb --- !passtrvb
                       --- !preposition --- jointermphrase >> drop3rd
```

Given the early history of the formal grammars, it is unsurprising that parser combinators are also related to research in natural-language processing (NLP). Parser combinators are often attributed to the paper of Frost and Launchbury [12], who used them to construct a natural-language processing tool in the Miranda language, an ancestor of Haskell. The picture above shows a fragment of their grammar, displaying a remarkable similarity to parboiled2 grammars. Frost and Launchbury used combinators, because of the compositional design and the ability to express rich semantic actions needed in NLP. Their syntax mimicked BNF, enabling fast prototyping and experimentation with language processors.

Parser combinators are used to build *recursive descent* parsers. Such parsers decide which production applies based on a prefix of the stream of symbols at the current position, and then invoke the production recursively. A production typically consumes the symbols from left to right and constructs the AST on the fly. Today, many mainstream parsing tools are recursive descent, as its semantics are easier for users to understand than the alternatives (say shift-reduce parsing).

Over time, parser combinator libraries became a part of the basic infrastructure of any serious programming language: Java, Scala (parboiled2 and Petit Parsers), JavaScript (Bennu, Parjs, Parsimmon), C# (pidgin, superpower, parseq), C++ (Cpp-peglib, boost meta-parse, boost-spirit, Parser-Combinators), Python (Parsec.py, Parsy, Pyparsing, parsita), and so on.

```
val Transition: (String, Option[String], String) => Pure.Transition =
  (input, output, target) =>
    Pure.Transition (target, input, output.getOrElse (""))
```
source: fsm.scala/src/main/scala/dsldesign/fsm/scala/FsmParser.scala

Figure 4.9: The semantic action for constructing instances of transition

example. This production gathers a string produced by three clauses (the middle one will be optional due to the question mark), and feeds them into a semantic action `Transition` shown in Fig. 4.9.

The action is a function taking the three constructed values and producing a transition object. The parsed values are just reordered, so that they match the order of arguments of the constructor (cf. Fig. 3.5), and the optional output is replaced with an empty string, if missing. This general way to specify semantic actions is far more expressive than the constructor calls and property assignments of Xtext. It almost never happens that one needs to adjust the grammar to allow the parser to easily construct the meta-model types when using parboiled2 (in Xtext we inlined rules because of this).

Overall, working with a general PEG parser and parser combinators gives us more possibilities than with fixed-formalism tools based on variants of context-free grammars, like Xtext and ANTLR. However, this flexibility comes at a non-trivial cost. First, the grammar specification in Xtext (for our example) is about three times shorter than in parboiled2. The production rules are similarly concise, but the need to explicitly write semantic actions, token parsing, and white-space handling creates a lot of additional work. Recall that Xtext provides a predefined library of tokens and a default white-space handling mechanism suitable for most needs. Also the semantic actions of Xtext, albeit limited, are introduced with minimal effort. Second, and perhaps more important, parser combinators are a powerful expressive tool, with much weaker error reporting than a closed-format Xtext editor. This translates to a much harder development experience. In the words of Ford [11], *a powerful syntax description paradigm also means more rope for the careless language designer to hang himself with*. Mistakes are harder to understand and debug. And one still typically needs to resolve named references afterwards, without any automatic support. It is clear that these two groups of tools represent very different strategies suitable for different use cases. External parsing tools are heavy dependencies for projects and require mastering a new grammar specification language. Parser combinators are a very lightweight dependency (just a library) that can be embedded in any place in an existing program written in the host language. They are particularly, but not only, suitable for small local parsing jobs. A language engineer, and indeed any experienced programmer, needs to be able to use either depending on the context.

4.5 Guidelines for Specifying Concrete Syntax

Let us switch from discussing concrete tools and case studies to general rules and guidelines for creating concrete syntax. We have identified a range of recommendations in research papers and through personal experience of teaching and developing DSLs. We begin with big decisions (whether to write a grammar at all!), and move through architecture-level guidelines (how to choose rules, how to modularize and reuse) all the way to low-level patterns (how to avoid left recursion, where to use grammars vs regular expressions, and how to handle comments).

Guideline 4.1 *Consider not writing a grammar. No parser at all!* Standard format technologies (YAML, JSON, XML, and CSV)[7] are natural alternatives to bespoke syntax. They allow fast and ad hoc creation of file formats with efficient parsers and validity checkers. These are excellent for many structural- and configuration-modeling tasks. On the other hand, bespoke syntax may be needed if humans have to create models in an editor, the DSL is complex, or the intended users do not have technical background. A tailor-made concrete syntax can also make users much more efficient, so consider it for high-volume tasks.

Another alternative to syntax design is to develop a GUI application for creating "models," typically a web-form or a wizard that populates a YAML/JSON/XML file, or stores data in a relational database. For many

[7] See also Sect. 3.10.

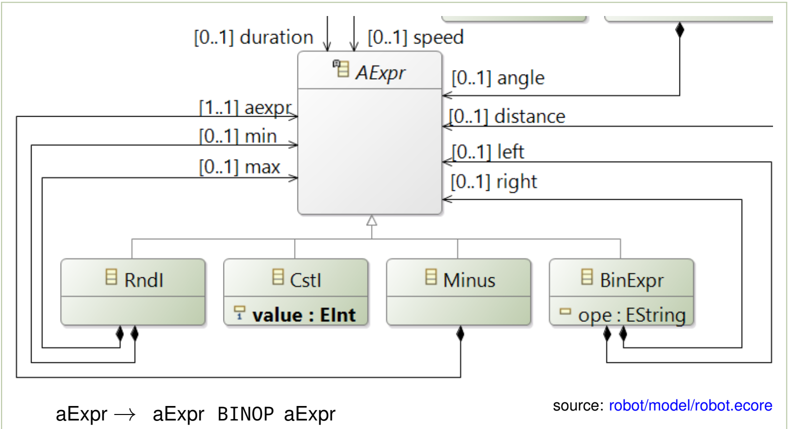
simple input formats, this will give a better user experience than a bespoke textual or graphical modeling syntax.

If the users of your language fall into several groups with distinct presentation requirements, it might be worthwhile to invest in creating multiple front-ends for the same abstract syntax, to allow the various tools to inter-operate with the back-end. For example, programmers and IT operations technicians can use textual syntax that resembles a programming language, while business product modelers would use a GUI or a graphical syntax, yet both would be producing and changing models in the same abstract language.

Textual or graphical syntax? Concrete syntax may be textual, graphical (typically some form of a diagram), or hybrid (for instance state-machine graphs with attached program code like in MATLAB/Simulink). The main advantages of the textual form are the clear order of reading and efficient typing with keyboards. Typically, mathematical expressions are hard to input with other means than a keyboard, so it is natural and efficient to express them as text. Furthermore, textual syntax is clearly the most popular among professional programmers—DSLs aimed at software developers should probably be textual [15]. Textual syntax is also the cheapest to implement, especially with language workbenches like Xtext, Monticore, or Spoofax. **Guideline 4.2**

On the other hand, textual syntax is relatively hard to read, especially for non-programmers. Text tends to hide indirections and references. While in graphical syntax edges (arrows, lines) can express relationships, textual syntax usually requires writing down an identifier as a reference to another element. The structure of complex relationships (beyond natural nesting, like partonomy) is obscured. For instance, it is very hard to spot a cycle or a bottom connected component in a finite-state machine expressed in the syntax of Fig. 4.5. If such structures must be visible, graphical syntax might be preferred. However, for complex and large files it is probably still better to replace reliance on visual skills with custom model analysis tools.

Use familiar, friendly, and intuitive notations, optimized for comprehension. **Guideline 4.3** It is well known that engineers are attracted to learning new languages, and that they tend to learn fast. It is quite the opposite for many non-programmers. A new notation is likely to become yet another barrier to adoption of the technology you want to introduce [14]. (Remember that your future users also need to learn the tools and to adapt to new work processes.) To minimize this risk, look to informal notations of the domain as the foundation for the DSL. Adopt whatever formal notations the domain experts already have and know, rather than invent new ones [14, 23]. Use their jargon terms whenever possible. Wile recalls a case where a concept called by experts a ‘metadata data item’ was not anything more than a ‘variable’ in the eyes of language designers. Still the original term, known to users, was kept in the language. For any non-fundamental issues, it is easier for language designers to adapt to users than the other way around. In another place, Wile reports sticking to an existing notation even if it



```
aExpr →  aExpr BINOP aExpr
aExpr →  '-' aExpr
aExpr →  'random' '(' aExpr ',' aExpr ')'
aExpr →  INT
```

Figure 4.10: A fragment of the `robot` *language meta-model presenting the abstract syntax of expressions (top). In the bottom, an ambiguous context-free grammar capturing the same syntax. See Chapter 2 for background about this example DSL*

appears bad from a programming language perspective; he recalls the case of a DSL where parentheses were not balanced.

For the same reason choose known symbols for known concepts. A plus ('+') should still mean addition and rarely anything else, etc. If in need of new symbols, use descriptive terms (English words) or multi-character symbols. In our `robot` language (see Chapter 2) we have a need to calculate angles, speeds, and durations. Such calculations do not differ essentially from basic mathematical (and programming) expressions. Thus we suggest using a basic abstract syntax and grammar for them as shown in the example for the `robot` language in Fig. 4.10. (Since core arithmetic expressions are extremely common, we return to them in the discussions below.) For the sake of readability, avoid overloading symbols, unless expected, and make different concepts visually distinct.

Models and programs are *read* much more often than *written* [14]. Hence, balancing comprehensibility and compactness is a delicate matter. For a designer it is fairly easy to focus on compact representations, but you should test your designs against users who try to *read* your example models.

Guideline 4.4 *Exploit the examples and the meta-model to structure grammar productions.* The least clear part of Sect. 4.3 is the selection of grammar productions. Let's dwell a bit longer on this problem. When writing a grammar, we are aiming at constructing a parse tree. What kind of branches do we have in the tree? The branches in the tree must agree with two sources of constraints: the input and the output structure.

We begin with seeking inspiration in the input. The most obvious structure of the production rules comes from nesting parenthetical constructions in your model. Look at a larger mock-up of your DSL syntax, squint your eyes, and observe the parenthetical structures: parts of syntax that

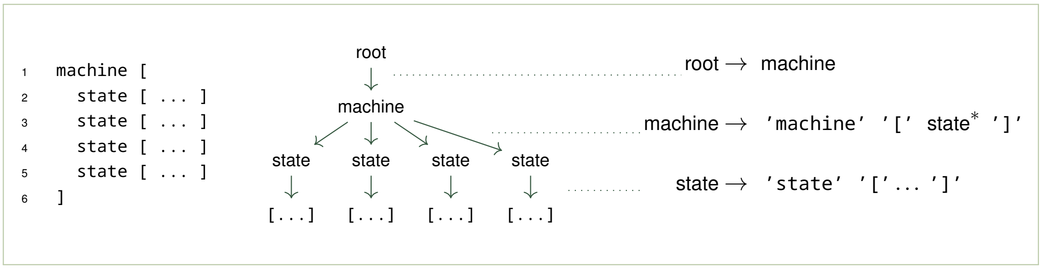


Figure 4.11: Picking up production nesting by parenthetical constructs. Left: the core nesting structure in Fig. 4.5, Center: a hypothetical core structure of a parse tree for the same example, Right: A hypothetical CFG able to generate the tree in the middle

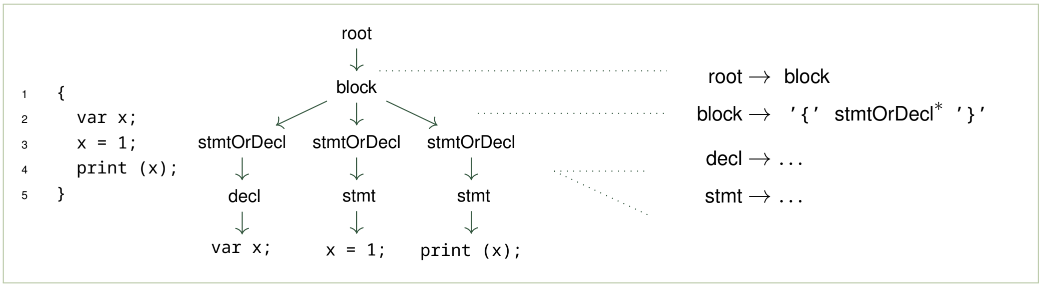


Figure 4.12: Creating an abstract nonterminal (stmtOrDecl) for elements appearing at the same level in a nesting structure of productions. The example uses a hypothetical Javascript-like syntax

are enclosed within fixed opening and closing elements. Some possible examples include: quotation marks, including double and triple quotation marks, keywords `begin...end`, tags `<div>...</div>`, funny keywords `if...fi`, `do...od` etc., as well as parentheses, brackets, braces, and so on. In languages like Python and Haskell, where white space is used to nest objects, the pairs could be indent/unindent, so they are a bit harder to see, but they are still there!

Figure 4.11 demonstrates nesting context-free productions according to nested structures in the input text. In the left of the figure, we show the model from Fig. 4.5, eliding non-parenthetical aspects to make the nesting stand out; in the middle, the same parenthetical structure is shown as a tree; in the right, we extract productions from the tree. Note how the direct nesting in the tree turns into productions (root and state). For machine we turn similar structures into a repetition mechanism (Kleene star), but otherwise the nesting still follows the tree. It is clear that the grammar on the right will generate trees like in the middle.

Figure 4.12 shows how to unify various syntactic structures that are allowed to be placed at the same nesting level. The left side includes a code fragment in a Javascript-like language. We have a block (delimited by braces) and, within this block, a handful of constructs that are syntactically different: the first one is a declaration, the last two are statements. Had you followed our advice from Sect. 4.3 you would have created two separate nonterminals for declarations and statements. What should we then nest under the block? Placing two or more syntactic categories at the same level is a common pattern. We would still like to handle these situations with simple replication like the machine/ state dependency in Fig. 4.11. To this end, we introduce a new abstract nonterminal (called stmtOrDecl here) that admits both kinds of expansions, or use the alternative operator, for example:

$$\mathsf{block} \to \ \texttt{'\{'} \ (\mathsf{stmt} \mid \mathsf{decl})^* \ \texttt{'\}'} \ . \tag{4.27}$$

Finally, the meaning of the parsed text often gives hints for the creation of rules: a coherent piece of syntax that returns a single value (an expression for instance) or performs a single coordinated action (a declaration of a complex type, and if-then-else statement, or a while-loop) are good candidates for grouping under a single rule. This is how we created the transition rule in Sect. 4.3. In Fig. 4.10, we chose a single non-terminal aExpr to group arithmetic expressions. A single top-level expression non-terminal will permit all kinds of expressions wherever an expression is needed in the `robot` language, resulting in a nicely orthogonal design.

Exercise 4.13. Consider the subset of Cascading Style Sheets (CSS) studied in Exercise 3.2 and in Fig. 3.2, p. 54. Write a simple context-free grammar in EBNF for this subset of CSS. Assume that the start nonterminal is called css. You need to decide what are the terminals in your language (typically keywords, operators, punctuation, and names), but you do not have to formally define them. Focus on the high-level structure, production nesting, and non-terminal selection.

Another, less obvious way to realize the structure of productions is to consider the output structure.[8] Since the parser creates an instances of a meta-model, the parse tree should be closely aligned with the main tree structure embedded in the meta-model. To appreciate this, revisit Fig. 4.10. The meta-model in the figure contains two kinds of lines: generalization/inheritance and containment relations. Observe that all the productions of the non-terminal aExpr (which constructs an instance of *AExpr*) follow the inheritance relations (the taxonomy tree); there is one production for each inheritance line. At the same time the containment relations (the partonomy) become references to nonterminals in the right-hand-side of the productions. This way a grammar can generate structures that can be typed by (can conform to) this meta-model.

There is a third kind of line we might see in a meta-model, the usual references (not shown in this figure). The non-containment references in the meta-model correspond to references to non-terminals in the right hand-side of productions, just like containment references. However, for non-containment references, we usually do not construct an instance of the sub-tree, but identify this sub-tree elsewhere, and link to it (reference it.) We have observed this mechanism in Line 25 of Fig. 4.6. Verify that the target property of a transition object in Fig. 3.1 is indeed a non-containment

[8]This is (roughly!) the procedure that the Xtext tool uses to generate an initial grammar for any given meta-model.

reference, and this is why we resolve a reference in the Xtext grammar there instead of constructing a new object.

If you use the target meta-model to construct a grammar, the grammar is likely to be ambiguous (many parse trees are possible for the same input) and left-recursive (left-biased recursive descent parsers will not terminate on it). Indeed, a grammar which simply captures abstract syntax will lack a few details needed to make it an effective parsing grammar. However, standard techniques (see below) can be used to eliminate the left recursion, and this, most often, will get rid of the ambiguity as well.

Exercise 4.14. Write a simple, possibly ambiguous and left-recursive, grammar for the language of feature models following the meta-model in Fig. 3.21 on p. 81.

Do not fight the input-output impedance in a grammar design. But what if the input structure and the output meta-model lead to a very different grammar? You are experiencing a case of an *input-output impedance*. Fighting an input-output impedance during parsing is usually a bad idea. A parser is not a suitable tool to mold the input data into an incompatible output structure. Parsing is difficult enough without this. It is better to keep the abstract and concrete syntax closer to each other. If you are experiencing an input-output impedance, design a new meta-model that is structurally similar to the input format. Populate this new abstract-syntax during parsing [14], and then use a separate transformation pass (outside the parser) to obtain an instance of the ultimate target meta-model. Parsing should to be compatible with abstract syntax, otherwise post-processing is needed. This post-processing is more easily done in a general programming language, after parsing. ***Guideline 4.5***

This is, in fact, what we did for the finite-state-machine language, when parsing with parboiled2 in Scala. Since the combinator-based parser was pure, it was difficult to create a cyclic graph structure instantiating the meta-model of Fig. 3.1. Instead, we used a simpler acyclic meta-model, which requires an additional transformation pass (see also Chapter 7).

Modularize your grammars [2]. Grammars for real languages can get large. Modularize your grammar *vertically* and *horizontally*, not only to help reuse in other language projects, but also to make it easier to understand and evolve your parser. For vertical modularization, group syntax elements in syntactic categories. Introduce non-terminals for entire categories and place productions defining the members of a category close to each other in a file. This kind of abstraction was proposed in Fig. 4.12, where we extracted stmtOrDecl, and in Eq. (4.23), where we extracted machineBlockContents. ***Guideline 4.6***

For horizontal modularization, split the definitions of tokens and the lexical structure from the high-level rules, even if your parsing system does not have a separate scanner, but uses context-free productions for the entire task. This is how we structured our parboiled2 grammar: the lexer in Fig. 4.8 and the “actual parser” in Fig. 4.7. Separating the lexical and syntactic productions has the additional advantage that it confines handling white space to the low-level rules (see below, p. 120).

Many grammar specification languages allow parts of AST definitions and grammar fragments to be imported. For instance, Xtext allows us to import terminal definitions and grammars, and multiple Ecore meta-models, so one can structure abstract syntax into several modules. In fact, an entire grammar for Java-like expressions (XBase) is provided. Above, we have imported the definitions of standard terminals (Line 2 in Fig. 4.6). Also when using parboiled2 you can split a large grammar into several modules. Use Scala/Java packages, imports, and generics to effectively compose them together. ANTLR,[9] Spoofax/SDF3,[10] TXL,[11] and most other language development systems today support import and modularity constructs.

Guideline 4.7 *Reuse existing grammars or parts of grammars.* As mentioned above, when using a rich language development system, or if you have modularized your previous grammars, you can reuse language design modules by importing. Instead of starting to design syntax from scratch, develop a habit to check what sub-languages have already been defined. You will save time on testing and getting things right. In the extreme, never plan to develop a grammar for a well-established language as the first line of attack. For most existing languages and sub-languages, open-source grammars are already available, either as part of their compilers or editing environments, or included in language workbenches as examples and resources.[12] Even if you cannot reuse the grammar directly due to different development languages and tools being used, you can quite often reuse the design, by transcribing the productions to your setup.

Guideline 4.8 *Handle white space at the lexer level.* In most situations, it is recommended to handle white space in a lexer. Most languages use the same treatment of white space (spaces, tabs, and newline characters). Dedicated lexers tend to have built-in support that does not require any specification—any white space is just ignored. It only separates tokens that could otherwise be confused, for instance adjacent tokens and identifiers: e.g., `state S0` should not be allowed to be written `stateS0`. Often it is possible to modify the definition of what characters count as white space, for the rare case when controlling them tightly is needed (see below).

The situation gets more complex when using a parser combinator library. PEGs are typically defined at the character level, like our example with parboiled2. There the programmer may match white space wherever she sees fit. Still, even with PEGs, it is a good practice to handle white space in the rules that logically belong to the lexer, so the productions building tokens. Anything else tends to lead to extremely complex and messy rule systems, which are hard to debug. Recall that in our Scala grammar, all white-space

[9] https://www.antlr.org/, retrieved 2022/09
[10] http://www.metaborg.org/en/latest/, retrieved 2022/09
[11] http://www.txl.ca, retrieved 2022/09
[12] See for example: the ANTLR grammar collection https://github.com/antlr/grammars-v4, TXL grammar collection https://www.txl.ca/txl-resources.html, and a large grammar zoo at the software language engineering body of knowledge website https://slebok.github.io/zoo/ (accessed 2022/09)

issues have been confined to the part shown in Fig. 4.8. The high-level productions in Fig. 4.7 remained purely at the syntactic level, disregarding individual character issues. Ford [11] recommends handling white space immediately after each token, and we mostly followed his advice in Fig. 4.8.

White-space-sensitive parsing. There are, of course, exceptions to the above rule. Some languages use indentation or line breaks as part of their syntactic structure. Hereunder Haskell and Python use indentation to mark code blocks (which many other languages solve with braces), and Scala allows line breaks to be used as statement separators (which most C-family languages, including Java, do with semicolons). White-space-sensitive parsing is a tempting design technique for DSLs as well: it allows us to create models that are more concise, and may resemble human-aimed notation. For instance, in the Clafer language [3], we chose to use indentation and newlines to group and separate model elements, so that models look more similar to notes made by someone collecting main bullet points about a domain. White-space-sensitive syntaxes also have some disadvantages; chiefly it is hard to move pieces of code between places at different levels in the blocks, and hard to reformat code automatically. They may also confuse programmers trained on traditional block-oriented syntax. ***Guideline 4.9***

When your language has white-space-sensitive syntax, the parser needs to consider white space. This is typically done by tracking the current nesting (counting tab characters or spaces), and using an explicit newline character as a separator in productions listing statements, declarations, etc. Unimportant white space, not at the beginning or at the end of a line, can still be handled at the lexer level for simplicity. Refer to the manual of your parsing system to see whether any explicit support is provided for white-space-sensitive parsing.

Allow comments and handle them at the lexer level. Always include some syntax for comments in your DSL. Comments not only allow users to annotate models, but also help them to experiment and to quickly hide defunct parts of the model. They are an important usability feature [14]. ***Guideline 4.10***

For most language implementation tasks, comments are considered white space and they should be handled in a lexer. Whether handled in a lexer or a parser, they are typically not saved in an AST, but just consumed. The only problem is caused by multi-line comments, which should be allowed to be nested for usability purposes, so that it is possible to comment out a piece of program that already contains comments. Nested comments, like nested parentheses, are not regular languages, so they cannot be defined just with regular expressions, without the additional power of context-free languages. In practice, lexers often include built-in extensions (for example nesting counters), so that it is possible to handle (even) nested comments at this level. Of course, PEGs, like in parboiled2, have no problems handling nested structures, so this is not an issue there. Ford [11] proposes the following rule for handling nested comments in a production. It uses a negative predicate (the exclamation mark that means “anything except” the operand).

$$\mathsf{comment} \rightarrow \quad \texttt{'/*'}\ (\mathsf{comment}\ \mid\ (!\ \texttt{'*/'}))^{*}\ \texttt{'*/'} \tag{4.28}$$

The rule says that a comment opens with a slash and asterisk and ends with an asterisk and slash. Between the delimiters, we allow an arbitrary mixture of other comments and any characters that are not a closing sequence for a comment (!'*/'). This rule could also be formulated in Xtext. The Xtext syntax specification language includes so-called until tokens, negative tokens, and hidden tokens[13] that help when parsing multi-line nested comments. The terminal grammar that we imported in our example supports default handling of Java-like multi-line and single-line comments using a simpler rule. This means that our implementation of the finite-state-machine language in Xtext allowed comments, although not nested comments.

Like for any other guideline, there is an exception to this one, too. If you are building a tool that processes comments, for instance a docbook/javadoc-style processor, or if the user comments are supposed to be forwarded to generated code, then comments need to be parsed for information with proper rules, and represented explicitly in the AST meta-model.

Guideline 4.11 *Do not use lexing and regular expressions for nested inductive structures.* Only use regular expressions for "finite memory" constructs. A common picture is a web-programmer trying to parse a complex input using regular expressions. The expressions grow and become increasingly complex, but some cases remain uncovered, and new bugs pop up all the time. We would like you to develop intuition for when to switch to grammars when parsing, so that you can avoid these frustrating situations in your developer practice.

Intuitively, a regular language can be recognized using a finite and bounded amount of memory. Languages that require counting during parsing are not regular. For instance the language representing all mixtures of balanced pairs of parentheses is not regular. Here is one example word in this language: "((((())))))." Imagine a finite-memory recognizer for this language as a finite-state automaton. With every open parenthesis we need to advance to a new state to remember how many are opened, and with each closed one we can retract to the previous state. So in the example above, we will advance through five states when opening the parentheses, and start to move back when the first one is closed. We can always create a sufficiently long string of nested balanced parentheses, on which your recognizer will "run out of memory" and loose track of balanced pairs during parsing. As soon as we have to reuse one of the previously visited states we will not know precisely how many parentheses have been opened: the number that was open at the very first visit, or at the second one. (Recall that for a finite automaton the only way to store information is to change states.)

This result is formalized in mathematical linguistics under the name the *pumping lemma for regular languages*. (We recommend the book by Hopcroft, Motwani, and Ullman [13] for a thorough study of this theory.)

[13] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#syntax, accessed 2022/09

It means that if there is some form of arbitrary nesting in your language, you will not be able to parse or validate it using regular expressions, but you need a grammar. In these cases, there is no point to "try harder" with regular expressions.

> **Exercise 4.15.** Recall that a polynomial is a function whose defining formula is a sum of terms; each term is a constant factor multiplied by a variable raised to a natural number. For example $2x^3$ is a term, and $2x^3 - 2y^2 + 7x$ is a polynomial. Consider the following grammar describing a language of simple polynomials, starting with the nonterminal poly.
>
> $$\begin{array}{llll} \mathsf{poly} \to_1 \mathsf{poly}\ \ \mathsf{sign}\ \ \mathsf{var}\ \ \texttt{'\^{}'}\ \ \mathsf{num} \mid \varepsilon & & \mathsf{var} \to_3 \texttt{'x'} \mid \texttt{'y'} \\ \mathsf{sign} \to_2 \texttt{'+'} \mid \texttt{'-'} & & \mathsf{num} \to_4 \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \end{array} \tag{4.29}$$
>
> In our polynomials, all terms must be signed for simplicity. First, write out one or two examples of polynomials generated by this grammar. Second, write a regular expression accepting the same language. Third, replace the production for var with: var → 'x' | 'y' | '(' poly ')'. Understand what new polynomials became syntactically legal; write one or two examples. Can we define a regular expression matching the language generated by the modified grammar?

Sometimes you just need to mix parsing and lexing. For some languages, it is not practical to separate parsing and lexing. This happens for some advanced (some would say "quirky") syntax designs. It may happen that interpreting a grouping of symbols into tokens depends on the parsing context. For instance, in C++ the sequence "<<" can be parsed as a single token (a shift-right arithmetic operator), or as two tokens (two opening angle brackets in a list instantiation). Compare how double angle is used in these two pieces of C++: "`x >> 2`" vs "`list<list<string>>`". In such situations, it is convenient to distinguish what tokens we are dealing with based on whether we are in the context of parsing a type expression, or an arithmetic expression. This is best done directly in the grammar productions, not in the lexer, when the high-level structure is not known yet. PEG parsing tends to support such cases well. **Guideline 4.12**

Interestingly, the C++ grammar, prior to version C++11, was defined with separate parsing and lexing phases; instantiating a list of lists of strings could not be written as above. Instead one should have written "`list<list<string> >`" separating the angle brackets, which is confusing for users. The newer versions of C++ and Java, which uses a similar syntax for generics, do not suffer from the same problem.

Even if you need to tokenize based on the syntactic context, we recommend limiting this practice to the absolute minimum, and still performing most tokenizing and parsing separately.

Avoid ambiguity in grammars. The grammar from the beginning of the chapter (4.14–4.16) is ambiguous. The rightmost derivation shown in (4.17) gives rise to the parse tree shown in Fig. 4.3. The following derivation, **Guideline 4.13**

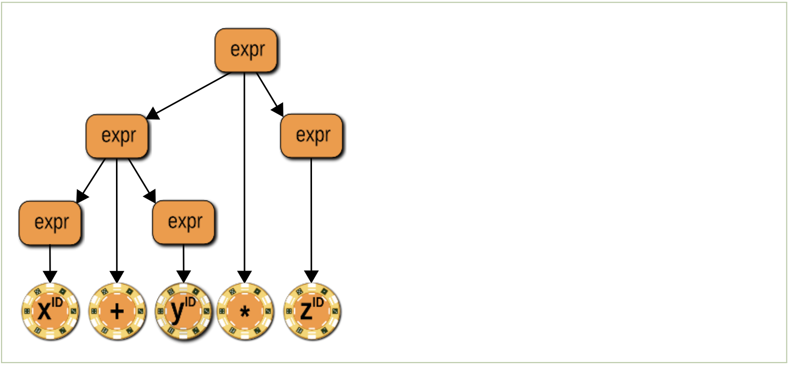


Figure 4.13: *The leftmost derivation tree for the expression $x+y*z$ using the grammar from Eqs. (4.14) to (4.16). Compare to Fig. 4.3*

which is also rightmost but picks the production rules in a different order, leads to the tree in Fig. 4.13. A different tree! Check!

$$\begin{aligned}
\texttt{expr} &\rightarrow_2 \texttt{expr '*' expr}\\
&\rightarrow_3 \texttt{expr '*' ID}\\
&\rightarrow_1 \texttt{expr '+' expr '*' ID}\\
&\rightarrow_3 \texttt{expr '+' ID '*' ID}\\
&\rightarrow_3 \texttt{ID '+' ID '*' ID} \qquad (4.30)
\end{aligned}$$

Exercise 4.16. Write down a *leftmost* derivation of the string $x+y*z$ using the grammar of Eqs. (4.14) to (4.16), and draw the corresponding parse tree. Which tree did you obtain? Is it the only possible leftmost derivation tree?

In general, we define ambiguity as follows.

Definition 4.31. *A grammar G is* ambiguous *iff there exists a word (a sequence of symbols) that can be derived from the start symbol of G in more than one way, so expanding nonterminals in a different order or using different productions, and resulting in two different parse trees.*

As you can see, depending on the order of applying the productions we obtain either a representation of $(x+y)*z$ or of $x+(y*z)$! (Which tree is which?) Not only for addition and multiplication, but in many other cases, this choice has serious consequences! You should control the ambiguity of your grammar so that you are sure that the precedence of the operators and similar structures is handled in agreement with your intentions.

For this very reason, many parsing tools restrict the input language for syntax specification to an unambiguous subset of context-free grammars such as LL(1), LALR, LL($*$), LR(k), or even PEGs. Typically, the ambiguity errors in input grammars are detected by these tools during parser construction. Most of these algorithms require that at any given time a rule can be chosen deterministically, otherwise an ambiguity is detected. PEGs eliminate non-determinism by using a fixed rule ordering combined with deterministic backtracking.

An ambiguity error message flags an error in your grammar, not in the parsing tool! Whatever tool your are using, you should understand whether it reports ambiguity errors, and what mechanisms it offers for handling ambiguity problems. Most parsing tools allow specification of the precedence of operators, which reduces non-determinism. Also, ambiguous grammars tend to be left-recursive, like our example with expressions. Eliminating left recursion tends to eliminate ambiguity as well (especially if the parsing tool follows a fixed left-, or right-parsing strategy). We talk about left-recursion elimination below.

Ambiguous grammars, like our expression grammar, tend to be easy to write. They strongly resemble abstract-syntax definitions. In fact, researchers often use ambiguous grammars to define "abstract syntax" in papers. If you are just starting to doodle a syntax for your language, it may well be easiest to start by proposing an ambiguous grammar first, a so-called baseline grammar, and to eliminate the ambiguities once you are satisfied with the core design.

Like every rule, also this one must have an exception. TXL [10] is a parsing tool that embraces ambiguity and expressly allows working with ambiguous grammars. This makes writing TXL grammars much easier, at the cost of making control over what trees are constructed more difficult.

Left-recursion elimination. The simple expression grammar used above is *left-recursive*. In production (4.16) the non-terminal expr is immediately expanded to another instance of expr, followed by some other symbols. Left-to-right parsers cannot handle left recursion, due to prefix-ambiguity. Let us try to understand why this might be a problem. Intuitively, left-to-right parsers try to match a rule like the one in (4.16), but cannot decide whether it is applicable or not. It seemingly allows infinite recursion: the very same rule can be tried immediately again and again. We define left recursion as follows. ***Guideline 4.14***

Definition 4.32. *A grammar is* left-recursive *if and only if it has a non-terminal symbol n such that there exists a derivation $n \rightarrow^{+} n\alpha$ for some arbitrary string of symbols α. [1]*

In other words, a grammar is left-recursive if it has a production with n on the left-hand side that can be expanded, possibly multiple times, until we obtain n as the leftmost symbol again. Productions (4.16) and (4.15) are both left-recursive. You are encouraged to convince yourself that none of the productions in Figures 4.6 and 4.7 are. This might be a bit harder to see in a grammar written using Xtext or parboiled2 than in abstract EBNF.

Inexperienced users of modern parsing tools frequently suffer from left-recursion issues. Only recently, ANTLR, which is the parsing tool underlying Xtext, started to support automatic left-recursion elimination for the special case of grammars with directly self-recursive productions (so all the left recursion appears in the same EBNF rule, perhaps using several alternative cases). At the time of writing however, Xtext does not benefit from this functionality. Thus Xtext grammars cannot be left-recursive, and ANTLR grammars cannot include left recursion along several separate productions. Furthermore, most PEG implementations, including parboiled2, simply loop indefinitely on left-recursive grammars.

This limitation of parboiled2, ANTLR, Xtext, and many other tools, is not a serious one, as it is widely believed that all interesting programming languages are specifiable in a non-left-recursive syntax. The only problem is that it sometimes takes some effort to put the grammar of the language in the right form. Learning how to do this also helps to fine-tune operator precedence and associativity, which is a useful skill, if you ever use a parsing tool without direct support for operator precedence specification.

Let us temporarily simplify our expression grammar to two rules, in order to facilitate explanation (we ignore the second rule, with the multiplication):

$$\mathsf{expr} \to \ \texttt{ID} \ \mid \ \mathsf{expr}\ \texttt{'+'}\ \mathsf{expr} \tag{4.33}$$

For brevity, we use parentheses instead of trees to show different parsings below. For the input string "`w+z+y+z`" the above grammar admits, among others, the following three parse trees:

$$\begin{array}{ll} ((w+x)+y)+z & \text{left-associative,} \\ (w+x)+(y+z) & \text{a balanced one,} \\ w+(x+(y+z)) & \text{right-associative.} \end{array}$$

If you cannot see why these trees arise, write out the corresponding derivations. To eliminate left recursion, we will disallow arbitrary parse trees, and focus on the last format from those listed above. The parsing $w+(x+(y+z))$ shows that a complex arithmetic summation is also just a sequence of additions, which starts with an identifier and then is followed by more identifiers separated by plus symbols. A plus-separated list of identifiers! A standard grammar generating a comma-separated list of identifiers is not left-recursive (Try to write it! see Exercise 4.34). We should be able to model a list of additions the same way.

In this optics, there is no inherent left recursion in long summations. The leftmost symbol in an input string is always a known terminal, here an `ID`, not a full-blown expression. Thus the following production:

$$\mathsf{expr} \to \ \texttt{ID}\ (\ \texttt{'+'}\ \texttt{ID})^* \ . \tag{4.34}$$

After this change no more left recursion remains, but we still express the same language as the original grammar. Let's generalize this example to a rule that handles the most cases of left recursion in practice. In the following figure, the grammar on the left can always be rewritten to the grammar on the right, without changing the generated language:

$$\boxed{n \to \beta \mid n\alpha} \quad \rightsquigarrow \quad \boxed{n \ \to \beta\ \ (\alpha)^*}$$

Figure 4.14: *The workhorse rewrite rule of the left-recursion elimination.*

In the figure, n is a non-terminal, β is a string of symbols not starting with n, and α is any string of symbols. For our example, $n =$ expr, $\beta =$ `ID`, and $\alpha =$ `'+' expr`.

The rule in Fig. 4.14 is slightly more general than what we did in our example. It keeps recursive expressions in α under Kleene-iteration, which is not a problem in this case, as they are not left-recursive. Admittedly, it is slightly hard to see that this rule may produce a right-heavy derivation tree, making the string α right-associative, as in $\beta(\alpha(\alpha(\alpha\cdots)))$. Appreciating this requires studying Tables 4.1 and 4.2 carefully. In practice, this also depends on how your parsing framework implements the Kleene star in EBNF. Most tools would just produce a flat list representation for parsing Kleene iterations.

Consider the original example again, where we had two inter-dependent left recursions. We recall it here for convenience:

$$\text{expr} \to \texttt{ID} \mid \text{expr}\ \texttt{'+'}\ \text{expr} \mid \text{expr}\ \texttt{'*'}\ \text{expr} \tag{4.35}$$

The rewrite rule from Fig. 4.14 does not apply directly anymore, as we have two cases of expressions. A naive attempt to generalize it could produce something like this:

$$\text{expr} \to \texttt{ID}\ (\ \texttt{'+'}\ \texttt{ID} \mid \texttt{'*'}\ \texttt{ID}\ )^* \qquad \text{(wrong!)}$$

The above production generates any mixture of multiplications and additions, which, in principle, means that we can handle all the strings we want. However, its derivation and parse trees disregard that the multiplication and addition have different precedence, so that multiplication should bind stronger than addition. For example, the string "`w*x*y+z`" may be parsed as $w*(x*(y+z))$ instead of the most likely desired $(w*(x*y))+z$. We will exploit the two precedence levels to remove left recursion here. At the top level we have addition, which binds weaker than multiplication. Our addition is still a plus-separated list, but the basic building blocks must be identifiers or multiplications of identifiers. We call these elements terms, as used in algebra for expressions that are summed. We apply the same trick as before to ensure that summations involve no left recursion:

$$\text{expr} \to \text{term}\ (\ \texttt{'+'}\ \text{term}\ )^* \tag{4.36}$$

$$\text{term} \to \text{term}\ \texttt{'*'}\ \text{term} \mid \texttt{ID} \tag{4.37}$$

We are not completely done yet! We still have left recursion in the second production (4.37), this time between terms. We have replaced expr with term, as we can only multiply identifiers and other terms. Multiplying expressions (so additions) is not possible, because addition has lower precedence. However, when doing this renaming, we have still allowed the second production to remain left-recursive. We shall apply the rewrite of Fig. 4.14 again to this rule, to eliminate the left recursion entirely:

$$\text{expr} \to \text{term}\ (\ \texttt{'+'}\ \text{term}\ )^* \tag{4.38}$$

$$\text{term} \to \texttt{ID}\ (\ \texttt{'*'}\ \texttt{ID})^* \tag{4.39}$$

In the new grammar, a summation term is an asterisk-separated list of identifiers. We obtained a grammar for expressions that accepts all the same inputs as the original example, but reconstructs the tree respecting the operator precedence. The key to get here was to apply the rewrite from Fig. 4.14 twice, once per each case, and also to observe that expressions with different precedence should be represented by different non-terminals (stratified), where we can use Kleene iteration at each level. The following figure summarizes the general rule for grammars with left recursion in multiple cases.

$$\boxed{n \to \beta \mid n\alpha n \mid n\gamma n} \quad \rightsquigarrow \quad \boxed{\begin{aligned} n &\to m\ (\alpha m)^* \\ m &\to \beta\ (\gamma\beta)^* \end{aligned}}$$

Figure 4.15: *The left-recursion elimination strategy with stratification of operator precedence.*

In the figure, n is a non-terminal and α, β, γ are any strings of symbols not containing n. We want α to bind weaker (have lower precedence) than γ. In our example, $n =$ expr, $\beta =$ `ID`, $\alpha =$ `'+'`, $\gamma =$ `'*'`, and $m =$ term.

Finally, what if we wanted to allow parentheses in our language, in order to override precedence? We leave understanding this issue to the reader by comparing the following two grammars. First, an ambiguous left-recursive grammar with parentheses:

$$\text{expr} \to \texttt{ID} \mid \texttt{'('}\ \text{expr}\ \texttt{')'} \mid \text{expr}\ \texttt{'+'}\ \text{expr} \mid \text{expr}\ \texttt{'*'}\ \text{expr} \tag{4.40}$$

and an unambiguous non-left-recursive grammar, where precedence has been enforced. (A factor is an expression that can be multiplied.)

$$\begin{aligned} \text{expr} &\to \text{term}\ (\ \texttt{'+'}\ \text{term}\ )^* \\ \text{term} &\to \text{factor}\ (\ \texttt{'*'}\ \texttt{factor})^* \\ \text{factor} &\to \texttt{ID} \mid \texttt{'('}\ \text{expr}\ \texttt{')'} \end{aligned} \tag{4.41}$$

Convince yourself that the two grammars generate the same strings, and that the second one is indeed not left-recursive, and that it creates derivation trees that respect the precedence of multiplication over addition, unless overwritten with parentheses.

4.6 Quality Assurance and Testing for Grammars

Focused tests for small grammar fragments. We strongly recommend to develop grammars iteratively. Do not attempt to write a grammar for a complex language in a single sitting. Even small grammars hide many intricate interacting constructs that are difficult to get right. Debugging a large grammar quickly becomes overwhelming. Instead, create, run, test, and fix coherent parts separately. At first, scaffold an empty parser that always fails, or always succeeds. Most tools support this with an empty start symbol production, or with a special “fail” (respectively “accept”) combinator. Make sure you can run your parser from this point on, every

```
"Transition variations (positive)" in new Fixture:
  """
    machine MACHINE [
      initial STATE
      state STATE [
        on input INPUT output OUTPUT and go to STATE
        on INPUT go to STATE
      ]
    ]
  """.parse[Model] should not be None

"A machine without initial state (negative)" in new Fixture:
  """
    machine MACHINE [
      state STATE []
    ]
  """.parse[Model] shouldBe None
```

source: fsm.xtext.scala/src/test/scala/dsldesign/fsm/xtext/scala/ParsersSpec.scala

Figure 4.16: *A positive and a negative test for the Xtext/ANTLR parser using the Xtext testing API, scripted in the Scalatest framework*

time you implement an extension or fix a bug. Build groups of productions bottom-up, starting from terminals, expressions, block-like compound groupings all the way to top-level concepts like modules, models, and programs. Feel free to ignore optional syntax elements in early iterations. Every time a meaningful subset of productions is specified, write a unit test for them, and keep these automated tests alive and passing throughout the development. Writing tests for small language fragments reduces the combinatorial explosion of testing on all possible input variations. It also gives you localized error information that is easy to interpret.

Do not stop working on a grammar when the parser works. Grammars should be optimized and refactored. Your first designs are likely to be suboptimal. One should eliminate excessive non-terminals and rules [2]. Optimization might give you a faster parser, but most importantly it helps you to understand your parser well. It helps you to spot and remove issues. It makes it easier for others to understand it, to remove any emerging problems, and to extend it in the future. This other person might be you in two years, surprised how complex a parser you made.

Positive and negative test cases. Figure 4.16 shows example tests for the Xtext parser of Fig. 4.6. These tests have been written using the Scalatest framework and the Xtext testing API. The testing framework and the programming language are inessential here. We could have written them using JUnit, or in any other JVM language, as these parsers are compatible with the standard JVM infrastructure. We want to draw your attention to (i) the format of the tests, and (ii) the use of positive and negative test cases. Regarding the format, when testing parsers you typically create small pieces of syntax (we are using Scala's multi-line strings here), then you *invoke the parser and inspect the result*. The `parse` method used in Fig. 4.17 is injected into the string class by the book library, which integrates Xtext with

```
"input, no output (positive)" in {
  "on input I go to T".transition.run () shouldBe
    Success (Transition ("T","I"))
}

"missing white space in transition (negative)" in {
  "onI goto T".transition.run ().toOption shouldBe empty
}
```
source: fsm.scala/src/test/scala/dsldesign/fsm/scala/FsmParserSpec.scala

Figure 4.17: *A positive and a negative test for the parboiled2 parser recognizing a transition, a part of the `fsm` language. The transition production is implemented in Fig. 4.7, Line 22.*

Scala to make writing Xtext tests in Scala more idiomatic.[14] The function returns `None` if the parser failed, and `Some` if it succeeded. In these two simple tests we only check for success, not for the structure of the created AST. This is often sufficient in small tests for DSLs.

We insist on *using both negative and positive test cases*. We should not forget that a parser fulfils two major roles: it translates an input to an abstract-syntax tree, which is later processed by other parts of the tool chain, and it validates the structure of the input. Testing a parser only on positive examples neglects its validation role. A good parser must fail on the erroneous input. Test syntactic constraints on examples that violate them, ideally *near-miss* examples that violate the rule but resemble a correct input. In the figure, the first test is positive, the second test is negative. Notice that the second input string looks like a plausible model—it takes some attention to notice that it lacks the initial state required by our syntax.

Figure 4.17 presents two test cases for the parboiled2 parser developed earlier in this chapter. The parboiled2 sub-parsers are accessible via a call to `run`. Here, `transition` refers directly to the transition production from Fig. 4.7. The direct access to sub-parsers is handy for testing parts of the grammar in the modular and incremental style recommended above. Most parser combinator libraries expose such an interface naturally, as all productions in these libraries are usually implemented using a single type—so every production can be used as a start production, a fully functional parser. In the first test, we not only check that it succeeded, but also that the created abstract-syntax tree value has the right structure.

At the time of writing, Xtext does not support testing parts of a grammar directly. This is why our Xtext tests invoked the top-level `Model` rule. An independent project provides facilities for production-level tests.[15]

Properties to test on grammars. When creating tests for parsers we recommend considering the following properties:

- *Handling white space.* For PEGs and any other parsers that mix lexing and parsing in a single mechanism, it is important to test whether white space is *allowed* where it should be, but it is *not required* more than strictly necessary. Arbitrary syntax errors involving spaces are irritating for users. Humans are not conscious of white space when reading—it is only important if its absence would cause confusion. For instance, in

[14] source: xtext.scala/src/main/scala/dsldesign/xtext/scala/package.scala

[15] https://github.com/itemis/xtext-testing, accessed 2022/09

Fig. 4.17 the second test establishes that white space is required between "`on`" and "`I`" if you want to interpret them as a keyword followed by an identifier—they are seen a single identifier otherwise. Dually, white space should not be required if tokens are clearly separatable visually, for instance between identifiers and operators, separators, or parentheses.

- *Optionality of elements.* Check whether the elements required to be optional can be omitted, and whether they can be added. Both errors are typical: you might have forgotten to include a question mark in an EBNF grammar, or to specify an entire optional clause.
- *Associativity and precedence of operators.* Test associativity if the order of evaluation influences the semantics of your operators. This is always the case if expressions have side effects. For operator precedence, compare ASTs both with and without parentheses, to check whether it is appropriately reflected in the nesting of the AST. These tests have additional importance if you use parser combinators. Parser generators (like Xtext/ANTLR) will warn you if you have left-recursion issues at generation time. Combinator parsers may enter an unbounded recursion at runtime, so it is good to test well at design time. See the discussion of left recursion, associativity, and precedence in Sect. 4.5.
- *Metamorphic properties.* Metamorphic properties are relations between data involved in several program runs. A classic metamorphic relation in parsing is that parsing an input, pretty-printing the resulting AST, parsing the pretty-printed output, and pretty-printing the obtained AST again, should produce the same AST and the same concrete syntax twice. This property can be tested on all valid inputs you have. Metamorphic relations are a good property to test if you have a lot of test cases, or if you have a possibility to generate inputs randomly. This avoids the problem of creating many test oracles manually, while it is still likely to find instabilities.

Test coverage for grammars. As always, the key coverage property to watch in testing is the *coverage of user requirements*. You should test whether user requirements are met. This is done by creating examples capturing the cases in the design and requirements documents (Sect. 3.2). At this stage, it is also useful to involve users. A few sessions with users, where you show them example models and ask them to create new ones, will uncover misconceptions in the syntax design, confusing notations, incomprehensible error messages, and missed requirements. Requirements can also be used to established parsimony of concrete syntax (cf. Def. 3.5). Since maintenance of DSLs is costly, we encourage you to look for nice-to-have but not required syntax extensions at this stage, and eliminate them from your grammar.

Finally, it is useful to ensure production and terminal coverage. This can often be done by creating one large input model including all features of the language [5]. In this chapter, the model in Fig. 4.5 was created to fulfil this role. The key advantage of this tactic is that it can be implemented very fast.

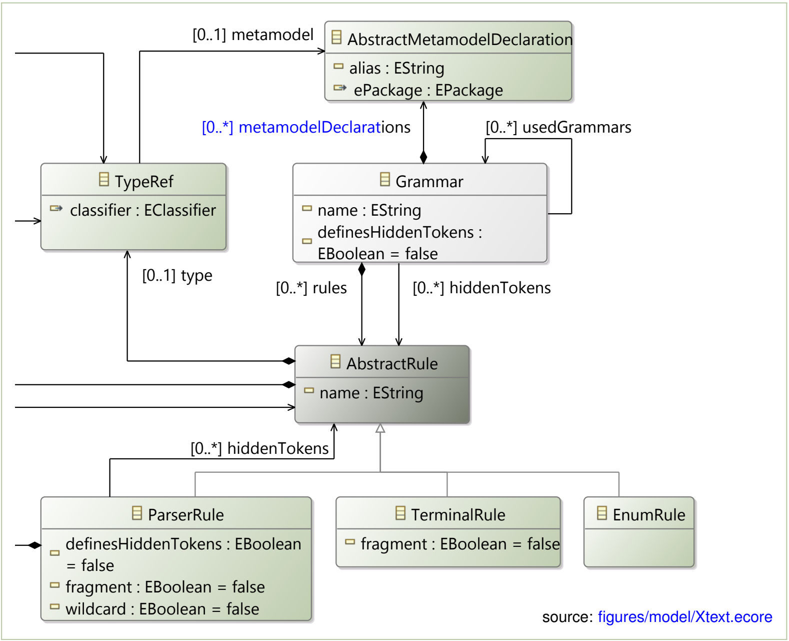

[0..1] metamodel
AbstractMetamodelDeclaration
alias : EString
ePackage : EPackage
[0..*] metamodelDeclarations
[0..*] usedGrammars
TypeRef
classifier : EClassifier
Grammar
name : EString
definesHiddenTokens : EBoolean = false
[0..1] type
[0..*] rules
[0..*] hiddenTokens
AbstractRule
name : EString
[0..*] hiddenTokens
ParserRule
definesHiddenTokens : EBoolean = false
fragment : EBoolean = false
wildcard : EBoolean = false
TerminalRule
fragment : EBoolean = false
EnumRule
source: figures/model/Xtext.ecore

Figure 4.18: A fragment of the meta-model used in the implementation of the Xtext framework for representing grammars

Exercise 4.17. Consider the following simple grammar for a subset of Cascading Style Sheets.[16] The non-terminal css is the start symbol. Devise a testing strategy for this grammar including test objectives, selection of test cases, scope of testing, and stopping criteria for the testing process. Show some example test cases.

$$\begin{aligned}
\textsf{css} &\rightarrow \textsf{specification}^{*}\\
\textsf{specification} &\rightarrow \textsf{element}\ \texttt{'\{'}\ \textsf{attribute}^{*}\ \texttt{'\}'}\\
\textsf{element} &\rightarrow \texttt{'p'} \mid \texttt{'div'}\\
\textsf{attribute} &\rightarrow \textsf{attrID}\ \texttt{':'}\ \texttt{color}\ \texttt{';'}\\
\textsf{color} &\rightarrow \texttt{'black'} \mid \texttt{'white'} \mid \texttt{'red'}\\
\textsf{attrID} &\rightarrow \texttt{'color'} \mid \texttt{'background-color'}
\end{aligned} \tag{4.42}$$

4.7 Grammars in the Language-Conformance Hierarchy

Let us step back for a moment and look at Figures 4.6 and 4.7 again. Both figures present language definitions. They define what models can be written in the finite-state-machine language. However, these language definitions are also models themselves. Yes! Grammars are models and parsers are programs. They are specified in a fixed specification language, a DSL with its own abstract and concrete syntax.

Since context-free grammars are a DSL, we can define abstract syntax (using meta-modeling) and grammars (using grammars!) for them. Fig-

[16] https://www.w3.org/Style/CSS/Overview.en.html, accessed 2022/09

ure 4.18 presents a fragment of the meta-model for the Xtext language.[17] Observe that the concepts in the meta-model reflect what we can present in a grammar, among others rules and tokens.

Exercise 4.18. Design a meta-model in Ecore (or an ADT in a functional language) for representing EBNF grammars as defined in Def. 4.18 and Tbl. 4.2. Inspect the Xtext meta-model linked above, to identify conceptual similarity.

Defining grammars for grammar languages is not an academic exercise in sophistry. It is yet another example of the design practice for language tools known as *bootstrapping*. Compiler builders for GPLs take implementing a compiler for their language as the first major project undertaken in the language itself; a rite of passage for the tools and the language design. This practice has spread from the compiler community to the broader community building language infrastructures in general. Thus Ecore models are represented as instances of the Ecore meta-model, which has been implemented in Ecore. Xtext grammars are parsed using an Xtext grammar, and so on. There are two important reasons for this practice. First, building a self-applying language tool is a rite of passage, the first major case study, for a language-processing system. If the designers of Xtext can "eat their own dog food" (use their own tool), then they can understand all the usability issues and develop it further in relevant directions. Second, the designers of language tools usually really believe in their ideas, so they are eager to use them, eager to demonstrate their usefulness. For designers of MDSE tools, like Xtext, the tools themselves serve as a major demonstration of the power of the paradigm. For example, it is thanks to the use of Xtext that the Xtext editor in Eclipse can offer syntax completion, and all the diagnostics facilities, based on just a small language definition.

Bootstrapping a compiler usually involves building an intermediary compiler in another language first, so that the first native-native compiler can be compiled. Similarly, bootstrapping a language-processing tool requires implementing a less powerful papier-mâché version of the tool. The early prototype should be powerful enough to build the first real bootstrapped tool. For instance, a parser generator might be first implemented using a manually built parser for its own grammar, or using a competing parsing tool. Once this works, we can generate the parser for the grammar specification language, and throw out the original simplistic manual implementation. Afterwards the tool can be evolved by itself, using its own infrastructure.

To make this discussion slightly more concrete, consider the problem of writing a grammar for regular expressions.

Exercise 4.19. Consider a standalone lexer generator (like Flex[18]). A lexer generator is a language-processing program. It reads a specification of a lexical structure (a set of named regular expressions defining tokens) and generates a piece of code,

[17]The meta-model is available in the source tree of Xtext, see https://github.com/eclipse/xtext-core/tree/master/org.eclipse.xtext/org/eclipse/xtext (seen 2020/09). Our code repository provides a laid out diagram, which was used to create Fig. 4.18. See figures/model/Xtext.aird

tokenizing a stream of symbols into a list of tokens. How is lexing done in a lexer generator? What are the tokens in the input for the lexer generator? Think about these questions before continuing to read.

The tokens in a regular expression language are (cf. Def. 4.12): a pipe (|), a plus (+), and an epsilon symbol (ε). We shall also add parentheses, to allow us to control the precedence, which was implicit in Def. 4.12. Having agreed on the tokens, we can write a grammar for regular expressions, so that we can parse them as part of a grammar definition, the lexical specification, for a hypothetical language-processing tool. If you look carefully, Def. 4.12 is already an ambiguous grammar in disguise. It is best if we reuse its structure:

$$\begin{array}{lll@{\qquad}lll}
\mathsf{regex} & \rightarrow & \mathsf{regex}\ \texttt{'|'}\ \mathsf{regex} & \mathsf{regex} & \rightarrow & \texttt{'('}\ \mathsf{regex}\ \texttt{')'} \\
\mathsf{regex} & \rightarrow & \mathsf{regex}\ \mathsf{regex} & \mathsf{regex} & \rightarrow & \texttt{'a'}\quad \text{for any character } \mathsf{a} \in \Sigma \\
\mathsf{regex} & \rightarrow & \mathsf{regex}\ \texttt{'+'} & \mathsf{regex} & \rightarrow & \varepsilon
\end{array} \tag{4.43}$$

Incidentally, the above grammar is ambiguous and left-recursive. Exercise 4.44 considers transforming it into a non-left-recursive form. Importantly, the pipe symbol above is a terminal symbol, not the alternative operator from EBNF (this is why we quoted it). Similarly the quoted plus symbol, and the quoted parentheses are not EBNF parentheses or Kleene iteration from EBNF. When writing grammars for a grammar DSL, we face the same cognitive confusion we experienced when discussing meta-models for meta-modeling languages in Chapter 3. The challenge there was that we had to use (meta-) classes to represent classes and objects. Now we are using the grammar rules to represent rules, and the same symbols appear possibly in two roles: as the object symbols and the meta-language symbols.

Exercise 4.20. Write an abstract EBNF grammar defining the syntax of the simplest context-free grammars, following Def. 4.18. Warning: this grammar will be extremely short, given how simple the syntax of grammar productions is.

Exercise 4.21. Expand the above grammar to generate the syntax of EBNF grammars. Your grammar should handle the EBNF operators as specified in Tbl. 4.2. Compare your grammar to the official Xtext grammar.[19] Are there any signs of conceptual proximity?

Figure 4.19 presents two of the top-level rules of the Xtext grammar for the Xtext language for comparison. Figure 4.20 summarizes the discussion of this section with a hierarchical diagram—a grammar counterpart of Fig. 3.13. In the bottom of the figure, we have the syntax of a concrete finite-state-machine model. This model is written in the Fsm language, so its syntax conforms to the `Fsm.xtext` grammar. Here conformance means that it parses without errors. The `Fsm.xtext` grammar is itself a model, written in the Xtext language, so it parses agains the `Xtext.xtext` grammar.

[18] https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator), seen 2022/09

[19] http://github.com/eclipse/xtext-core/blob/master/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext

```
Grammar:
  'grammar' name=GrammarID
    ('with' usedGrammars+=[Grammar|GrammarID]
      (',' usedGrammars+=[Grammar|GrammarID])*)?
    (definesHiddenTokens?='hidden'
    '(' (hiddenTokens+=[AbstractRule|RuleID]
       (',' hiddenTokens+=[AbstractRule|RuleID])*)? ')')?
    metamodelDeclarations+=AbstractMetamodelDeclaration*
    (rules+=AbstractRule)+ ;

AbstractRule: ParserRule | TerminalRule | EnumRule;
```

source: github.com/eclipse/xtext-core/blob/master/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext

Figure 4.19: *The top-level production (Grammar) of the grammar for the Xtext input format (describing grammars)*

Because of bootstrapping, the Xtext grammar is specified in itself, and is possible to parse against `Xtext.xtext`. This actually happens when you compile Xtext from source. In the right-hand side of the figure, we list example files in the languages listed to the left. You will notice that all these examples have been used earlier in the chapter to present these languages.

Further Reading

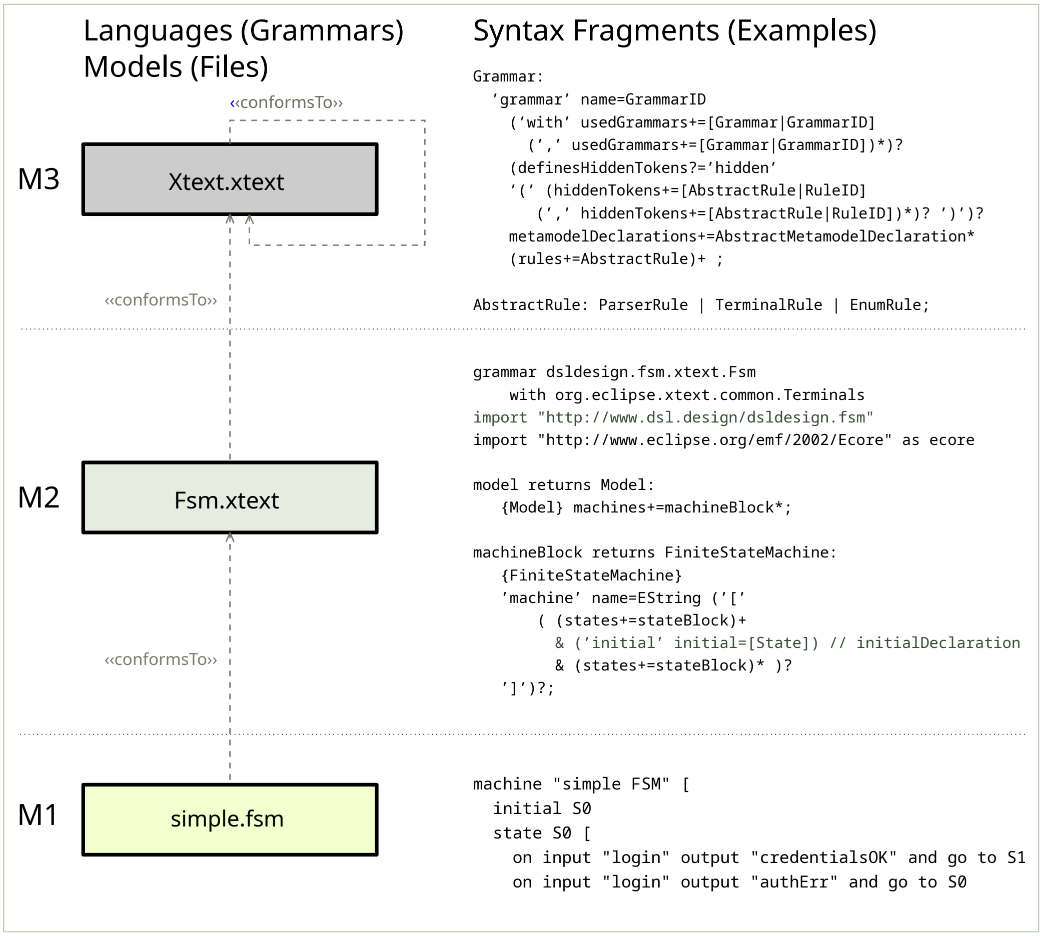
The standard reference on grammars and parsing is the *Dragon Book* by Aho et al. [1]. However, many competing books exist and most of them are very good. A more recent concise reference has been authored by Mogensen [21]. Classic compiler books have the advantage that they discuss different categories of grammars and different classes of parsing algorithm with varying strengths and weaknesses; a nerdy zoo of exotic constructions, mostly irrelevant for an average DSL designer. Thus we limited ourselves to a rather superficial discussion of parsing issues. Anybody building a parsing tool or experiencing performance issues with a parser (a relatively rare situation with DSLs), is encouraged to delve deeper into the subject, starting with the above two volumes.

The documentation of parboiled2[20] is helpful if you need to learn to use the combinators. Myltsev [22] describes the design principles and the implementation of the parboiled2 tool. Chiusano and Bjarnason [8] devote a chapter to the case study of a design of a parser combinator library in Scala (Chapter 9 therein). Interestingly, as of today, the problem whether PEGs and CFGs are incomparable is still open. Ford [11] shows languages accepted by a PEG that cannot be generated by any context-free grammar. However, we still do not know whether there exist context-free languages that are not possible to accept with a PEG. Recently, Loff, Moreira, and Reis [19] show that PEGs are surprisingly expressive, which is an indication (not a proof yet) that they might be a strictly more expressive formalism than CFGs.

The problem of checking whether a given context-free grammar is ambiguous is undecidable in general. Knuth [17] was probably the first to propose a conservative procedure for deciding the problem, based on detecting LR(k) shift–reduce conflicts. More recently, Brabrand, Giegerich, and Møller [7] give a short account of the state of the art on grammar ambiguity checking, and give a heuristic conservative procedure for detecting ambiguity.

You might be surprised to see that the recursive descent LL-parsing using left-recursive grammars is a solved problem, at least theoretically. Unfortunately, GLL

[20] Available at their GitHub page https://github.com/sirthias/parboiled2, seen 2022/09



```
Grammar:
  'grammar' name=GrammarID
    ('with' usedGrammars+=[Grammar|GrammarID]
      (',' usedGrammars+=[Grammar|GrammarID])*)?
    (definesHiddenTokens?='hidden'
    '(' (hiddenTokens+=[AbstractRule|RuleID]
       (',' hiddenTokens+=[AbstractRule|RuleID])*)? ')')?
    metamodelDeclarations+=AbstractMetamodelDeclaration*
    (rules+=AbstractRule)+ ;

AbstractRule: ParserRule | TerminalRule | EnumRule;
```

```
grammar dsldesign.fsm.xtext.Fsm
    with org.eclipse.xtext.common.Terminals
import "http://www.dsl.design/dsldesign.fsm"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

model returns Model:
   {Model} machines+=machineBlock*;

machineBlock returns FiniteStateMachine:
   {FiniteStateMachine}
   'machine' name=EString ('['
       ( (states+=stateBlock)+
         & ('initial' initial=[State]) // initialDeclaration
         & (states+=stateBlock)* )?
   ']')?;
```

```
machine "simple FSM" [
  initial S0
  state S0 [
    on input "login" output "credentialsOK" and go to S1
    on input "login" output "authErr" and go to S0
```

Figure 4.20: *A hierarchy of concrete syntax languages (with the finite-state-machine language in the bottom). Compare to Fig. 3.13*

parsing methods [18] are entering mainstream tools extremely slowly. Open-source libraries are only starting to appear and gather initial interest.[21] Independently of the developments in generalized parsing, there are many works on automatic and manual left-recursion elimination. Above, we presented a simple method loosely inspired by the section on left-recursion elimination in the book of Aho et al. [1]. Medeiros, Mascarenhas, and Ierusalimschy [20] propose a method to systematically and automatically handle left-recursive PEGs. This method has not been implemented in parboiled2 at the time of writing.

Xtext comes with extensive documentation for language designers.[22] Bettini [6] has written a book on the framework, the best reading material on the topic so far. It also explains left-recursion elimination for Xtext grammar files in detail.

[21] Examples: https://github.com/rust-lang/gll for Rust, https://github.com/djspiewak/gll-combinators for Scala, seen 2022/09

[22] http://www.eclipse.org/Xtext/, seen 2022/09

Additional Exercises

Exercise 4.22. The regular expression ab^* describes the set of all words starting with a symbol a and followed by zero or more bs. The expression $(aa)^+$ describes the language of all non-empty words that can be built from symbol a that have even length. Explain in English what are the languages described by the following expressions: **a)** $(10)^*$, **b)** $1(0|1)^*$, **c)** $(\ (0(1|2)3)_\sqcup)^+$. Parentheses are meta-operators used for grouping, and $\sqcup$ denotes a single blank character.

Exercise 4.23. Decide whether each of the following strings belongs (or not) to the language generated by the regular expression: `'0'|['0'-'9']+'.'['0'-'9' 'a'-'f']*`. Explain why. **a)** `'c0ffee.0730'`, **b)** `'0'` **c)** `'1'` **d)** `'0830.c0ffee'` **e)** `'09ea67.'` .

Exercise 4.24. The language L comprises words consisting of zero or more repetitions of a followed by a single b or a single c. If the final symbol is b the number of as must be even. If the final symbol is c the number of as may be even or odd. Write a regular expression matching/generating the language L.

$$L = \{b, aab, aaaab, aaaaaab, \ldots\} \cup \{c, ac, aac, aaac, aaaac, \ldots\} \qquad (4.44)$$

Exercise 4.25. Write a regular expression specifying identifiers as in the following quote from the ISO C standard: *An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter.* We recommend writing out several positive and negative examples first.

Exercise 4.26. The following regular expression defines a language of identifiers built from small letters and underscores: `('_'|['a'-'z'])*`. Improve it so that an identifier can no longer be built solely of underscores (if it starts with underscore it has to contain some letters, and possibly more underscores mixed in).

Exercise 4.27. Write a regular expression capturing unsigned fixed-point numbers with up to three digits precision. There are no restrictions on the leftmost and the rightmost zeros. There must be at least one digit to the left and at least one to the right of the decimal point. Positive examples: 1.5, 123456.00, 199.159, 001.1; Negative examples: 7, 5.000001, .99

Exercise 4.28. Write a regular expression matching hexadecimal numbers.

Exercise 4.29. The following regular expression matches fixed-point decimal constants: `['0'-'9']+'.'['0'-'9']+`. **a)** Show an example of a string that begins with a zero and matches this expression. **b)** Show a string that *ends* with zero and matches. **c)** Modify the expression to disallow prefix zeros and trailing zeros after the decimal point, except if a zero is the only symbol before or after the point.

Exercise 4.30. Write a regular expression matching a correct cardinality expression of the Clafer language [3], according to the following specification: A *cardinality expression* is enclosed in square brackets and consists of two integer constants separated by two consecutive dots. For instance: ‘`[1..0]`’, ‘`[5..10]`’ and ‘`[11..00]`’, but not ‘`[0..1..2]`’. See also Exercise 4.47

Exercise 4.31. Write a regular expression (grammar) generating (parsing) Roman numerals up to 100. Your expression should only match valid numerals, not just any combination of letters used in them.

Exercise 4.32. Explain in English what is the language described by the following context-free grammar, with s being the start symbol:

$$\begin{array}{llll} \texttt{s} \to_1 \texttt{t u 'a' v} & & \texttt{u} \to_3 \texttt{'reads'} \mid \texttt{'writes'} \\ \texttt{t} \to_2 \texttt{'John'} \mid \texttt{'Mary'} \mid \texttt{'Alice'} & & \texttt{v} \to_4 \texttt{'book'} \mid \texttt{'letter'} \mid \texttt{'poem'} \end{array}$$

Exercise 4.33. Explain in English what language is generated by the following EBNF grammar, with s being the start symbol.

$$\texttt{s} \to_1 \texttt{s op id} \mid \texttt{id} \qquad \texttt{op} \to_2 \texttt{'->'} \mid \texttt{'.'} \qquad \texttt{id} \to_3 \texttt{'x'}$$

Exercise 4.34. This context-free grammar accepts comma-separated lists of identifiers. The start symbol is s and `ID` refers to a standard Java identifier token.

$$\texttt{s} \to_1 \texttt{'(' t ')'} \qquad \texttt{t} \to_2 \texttt{ID ',' t} \qquad \texttt{t} \to_3 \varepsilon \qquad (4.45)$$

a) Does the string `(a, b, c)` belong to the language generated by this grammar?
b) If yes, show a derivation. If not, fix this grammar so that it belongs to it.
c) Write a regular expression that accepts the same language as the grammar.

Exercise 4.35. Specify concrete syntax for a comma-separated list of hexadecimal numbers. Each number is built of one or more white-space-separated groups of digits. Each group consists of four digits, except for the leftmost (most significant) group which can contain fewer. Decide whether to use regular expressions, grammars, or both to solve the task, and argue for your choice. *A positive example:* `c0 ffee, ff, f10 abcd 0123'`. *A negative example:* `'c0ff ee, abcd0123'`.

Exercise 4.36. In the following context-free grammar, s is the start symbol. Write a regular expression that accepts the same language.

$$\texttt{s} \to \texttt{a b c} \qquad \texttt{b} \to \texttt{b '1'} \mid \varepsilon \qquad \texttt{a} \to \texttt{a '2'} \mid \varepsilon \qquad \texttt{c} \to \texttt{c '3'} \mid \varepsilon$$

Exercise 4.37. Write a grammar representing the language of balanced parentheses of three kinds, so `'('`, `'{'`, and `'['`, where they can be arbitrarily nested as long as they are always balanced with a closing parenthesis of the same kind. A positive example: `(())[{}]`, a negative example: `([){]}`.

Exercise 4.38. Recall the language L from Exercise 4.24. Write a context-free grammar in EBNF generating this language, replacing the original regular expression. The grammar may be ambiguous and left-recursive. Symbols `'a'`, `'b'`, and `'c'` will be terminals in your grammar.

Exercise 4.39. Show *two different* derivations of *different length* of two strings from the following grammar (s is the start symbol). Mark the derivation arrows with production numbers, so that it is easy to reconstruct the rule application order.

$$\begin{array}{llll} \texttt{s} \to_1 \texttt{'a' 'b' s} & \texttt{s} \to_3 \texttt{'(' s ')'} & \texttt{s} \to_5 \texttt{'d'} & \\ \texttt{s} \to_2 \texttt{'g'} & \texttt{s} \to_4 \texttt{'a' 'b' s} & & (4.46) \end{array}$$

Exercise 4.40. In the following grammar, the start symbol is start. Is this grammar left-recursive? If not, explain why. If yes, eliminate the left recursion (write down the non-left-recursive grammar in EBNF accepting the same language).

$$\begin{aligned} \text{start} &\rightarrow \texttt{'('}\ \text{parameterList}\ \texttt{')'} \\ \text{parameterList} &\rightarrow \text{parameter} \mid \text{parameterList}\ \texttt{','}\ \text{parameter} \\ \text{parameter} &\rightarrow \texttt{ID ID} \end{aligned} \tag{4.47}$$

Exercise 4.41. Which of the following grammars are left-recursive? Symbol s is the start symbol.

a) $\texttt{s} \rightarrow_1 \texttt{s g}$, $\texttt{g} \rightarrow_2 \texttt{'a' 'b'}$, $\texttt{s} \rightarrow_3 \texttt{'c' 'd'}$

b) $\texttt{s} \rightarrow_1 \texttt{g s}$, $\texttt{g} \rightarrow_2 \texttt{'a' 'b'}$, $\texttt{s} \rightarrow_3 \texttt{'c' 'd'}$

c) $\texttt{s} \rightarrow_1 \texttt{x y z}$, $\texttt{x} \rightarrow_2 \texttt{z}$, $\texttt{z} \rightarrow_3 \texttt{'a'} \mid \texttt{'b'} \mid \texttt{s}$, $\texttt{y} \rightarrow_4 \texttt{'c'} \mid \texttt{'d'}$

Exercise 4.42. Eliminate left recursion from the following grammars:

a) $\text{stmt} \rightarrow_1 \text{stmt}\ \texttt{';'}\text{stmt}$, $\text{stmt} \rightarrow_2 \texttt{'\{'}\text{stmt}\ \texttt{'\}'}$, $\text{stmt} \rightarrow_3 \texttt{'print'} \mid \texttt{'skip'}$

b) $\text{qualified-name} \rightarrow_1 \text{qualified-name}\ \texttt{'.'}\ \texttt{'ID'}$, $\text{qualified-name} \rightarrow_2 \texttt{ID}$

Exercise 4.43. Consider the following definition of the *conjunctive normal form* (CNF) for propositional logic formulae: A *literal* is a variable identifier. An *atom* is either a literal (say x) or a negation of a literal (say $\neg x$). A *clause* is a disjunction of several atoms (possibly zero), for example: $(x \mid\mid y \mid\mid \neg z)$. A CNF *formula* is a conjunction (&&) of zero or more clauses. Write a non-left-recursive EBNF grammar for parsing propositional formulae in CNF, as defined above. Your grammar, should be able to parse, among others, the following example: $(x \mid\mid y \mid\mid \neg z)$ && $(\neg x)$ && $(x \mid\mid \neg x)$.

Exercise 4.44. Eliminate left recursion from the grammar in Eq. (4.43) on p. 134.

Exercise 4.45. This exercise attempts to develop a more domain-specific syntax for Morse code than the one presented in the chapter. A message in Morse code consists of a sequence of short and long tones. We can represent a short tone by a single dash character (`-`, a minus) and a long tone by three consecutive dashes without any spaces between them (`---`). Spaces separate long and short tones. A single slash character (`/`) marks a break between words. Write a short valid input string in this syntax. For instance, transcribe "MDSE IS FUN". Write an EBNF grammar defining a message in the Morse code over the set of the above four tokens (long tone, short tone, space, and slash).

Exercise 4.46. Revisit the mathematically oriented syntax for finite-state machines presented in the left part of Fig. 4.4. Write two or three more variants of this example. For instance, consider whether it is required to use let definitions, or whether the definitions can be nested directly in the "simpleFSM," whether naming of finite-state machines can be optional? Then write an abstract EBNF grammar generating this language or implement it in your favorite syntax specification tool.

Exercise 4.47. Parsing cardinality expressions using a regular expression is suboptimal (cf. Exercise 4.30). Separating elements of the expression is clumsy, and it is messy to control the white space. Write an EBNF grammar for Clafer's cardinality expression, to meet the following slightly richer specification than above.

A *cardinality constraint* is enclosed in square brackets and consists of two integer constants separated by two consecutive dots. For instance: '`[1..0]`', '`[5..10]`' and '`[11..00]`', but not '`[0..1..2]`'. A cardinality constraint can also be a single character selected from `'?'`, `'+'`, `'*'`. Assume that there exists a terminal symbol `INT` that is defined, and you can use it in your grammar. It matches non-negative integer constants.

Exercise 4.48. [mini-project] The advantage of a rich language workbench (Xtext) over a simple parser (parboiled2) is that it can support a broader range of use cases than just parsing. Use editor generation facilities to generate an Eclipse plugin for the finite-state-machine language, and to generate a web editor for this language. Use the Xtext documentation for detailed steps in the process.

Exercise 4.49. [mini-project] Use https://github.com/xtext/xtext-external-editors to generate vim, atom, and sublime syntax definitions for your external DSL. Alternatively, develop your own generator of syntax-highlighting from Xtext grammars. Use the tool to obtain syntax highlighting models for complex Xtext languages (for example Xtend and Xtext itself).

Exercise 4.50. [mini-project] Use the Xtext New Project wizard to initialize a grammar from an existing Ecore meta-model. Use the meta-model for feature diagrams shown in Fig. 3.20 on p. 81 available from the book code repository at featuremodels/model/FeatureModels1.ecore. Xtext will generate a default grammar for this language. Edit the generated grammar to improve readability and write-ability of the syntax. Revise the syntax and test in a generated editor as many times as you need, until you are satisfied.

Exercise 4.51. Recall that, unlike context-free grammars, program expression grammars are deterministically processed from left to right, and once a rule matches, a typical PEG parser does not backtrack. Consider a variant of the rule for `inputClause` from Equation (4.21). We basically replace the optionality by an alternative. Interpret the following production as a PEG not a CFG rule:

$$\text{inputClause} \rightarrow \texttt{'on'}\ \texttt{'input'}\ \texttt{ID} \mid \texttt{'on'}\ \texttt{ID} \tag{4.48}$$

Does reordering (swapping) the two operands of the alternative in the above production affect the language this production accepts? *Reflection points:* Ford [11] writes that this question is often obvious, but sometimes gets difficult. In general, it is an undecidable problem. This problem is trivial for context-free grammars—the reordering never changes the generated language. (Think why!) There is an interesting duality between PEGs and CFGs: for PEGs ambiguity is trivial (always unambiguous) but commutativity of alternative is undecidable. For CFGs the alternative operator is commutative, while ambiguity is undecidable.

Exercise 4.52. Following Ford [11], parboiled2 supports *syntactic predicates*. A syntactic predicate enforces a condition on the current symbol. A positive predicate (*must hold*) is written `&(p)` and a negative predicate (*must not hold*) is written `!(p)`. The predicate action does not consume any symbols from the input, and does not add anything to the output. The rule just fails and backtracks if a predicate is violated. The predicate `p`, in great simplicity can be any Boolean function that examines the current symbol, for instance: Is it a digit? Is it a letter? Is it capitalized? Typically with PEGs, the symbols are input stream characters.[23]

Rewrite the `ID` production in Fig. 4.8 to use only `IDSuffix` and a negative predicate instead of `IDFirst`. *Notes:* The formulation in Fig. 3.5 is likely better, but the point is to practice the use of predicates in PEGs. If you seek an example, Ford [11] shows a negative predicate in Figure 1, the `Primary` rule.

Exercise 4.53. Design positive and negative test cases for the grammar of Equation (4.41). Select the test cases to ensure good coverage, and argue for your selection of cases, as well as for the choice of the coverage metric.

Exercise 4.54. Design a concrete textual syntax for simple Ecore-like models, including classes, binary references, and generalization. Express your syntax definition as a context-free grammar.

Exercise 4.55. Design a grammar for the core of the XML language (opening/closing tags, attributes, and standalone empty-element tags). Make the exercise more challenging by including arbitrary non-tag strings inside the elements, possibly using negative syntactic predicates.

Exercise 4.56. Recall the basic syntax of grammars without any EBNF extensions and parentheses, so as shown in Def. 4.18. This syntax is so simple that it can be described using a regular expression (sic!). Write this regular expression matching a valid grammar production. Assume that the expression is written over the tokens: `ID` and `->`. These tokens are obviously also regular, so they can be inlined into your solution without losing the regularity of the language.

Exercise 4.57. This exercise can be solved after reading Chapter 5. Use your favorite parsing tool to define a grammar and parse the Alloy instance syntax, as shown in the right panel of Fig. 5.14 on p. 173. See also Exercise 3.34 on p. 84.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Prentice Hall, 2006 (cit. pp. 125, 135, 136).

[2] Tiago L. Alves and Joost Visser. “A case study in grammar engineering”. In: *SLE*. Vol. 5452. Lecture Notes in Computer Science. Springer, 2008 (cit. pp. 119, 129).

[3] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. “Clafer: Unifying class and feature modeling”. In: *Software and System Modeling* 15.3 (2016) (cit. pp. 121, 137).

[4] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. “Editing support for software languages: Implementation practices in language server protocols”. In: *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2022 (cit. p. 107).

[5] Jon Bentley. “Programming pearls: Little languages”. In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721 (cit. p. 131).

[6] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt, 2013 (cit. pp. 109, 136).

[7] Claus Brabrand, Robert Giegerich, and Anders Møller. “Analyzing ambiguity of context-free grammars”. In: *Sci. Comput. Program.* 75.3 (2010), pp. 176–191 (cit. p. 135).

[8] Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. Manning, 2014 (cit. p. 135).

[9] Noam Chomsky. *Syntactic Structures*. Mouton & Co., 1957 (cit. p. 97).

[23] More about predicates in parboiled2 grammars at https://github.com/sirthias/parboiled2.

[10] James R. Cordy. “The TXL source transformation language”. In: *Sci. Comput. Program.* 61.3 (2006), pp. 190–210 (cit. p. 125).

[11] Bryan Ford. “Parsing expression grammars: A recognition-based syntactic foundation”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by Neil D. Jones and Xavier Leroy. POPL. ACM, 2004 (cit. pp. 110, 114, 121, 135, 140).

[12] R. Frost and John Launchbury. “Constructing natural language interpreters in a lazy functional language”. In: *Comput. J.* 32.2 (1989), pp. 108–121 (cit. p. 113).

[13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001 (cit. pp. 93, 122).

[14] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. “Design guidelines for domain specific languages”. In: *9th OOPSLA Workshop on Domain-Specific Modeling*. 2009 (cit. pp. 115, 116, 119, 121).

[15] Steven Kelly and Risto Pohjonen. “Worst practices for domain-specific modeling”. In: *IEEE Software* 26.4 (2009), pp. 22–29 (cit. p. 115).

[16] Paul Klint, Ralf Lämmel, and Chris Verhoef. “Toward an engineering discipline for Grammarware”. In: *ACM Trans. Softw. Eng. Methodol.* 14.3 (2005), pp. 331–380 (cit. pp. 87, 99).

[17] Donald E. Knuth. “On the translation of languages from left to right”. In: *Information and Control* 8.6 (1965), pp. 607–639 (cit. p. 135).

[18] Bernard Lang. “Deterministic techniques for efficient non-deterministic parsers”. In: *Automata, Languages and Programming, 2nd Colloquium, Proceedings*. Ed. by Jacques Loeckx. Vol. 14. Lecture Notes in Computer Science. Springer, 1974 (cit. p. 136).

[19] Bruno Loff, Nelma Moreira, and Rogério Reis. “The computational power of parsing expression grammars”. In: *Developments in Language Theory - 22nd International Conference*. Ed. by Mizuho Hoshi and Shinnosuke Seki. Vol. 11088. Lecture Notes in Computer Science. Springer, 2018 (cit. p. 135).

[20] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. “Left recursion in parsing expression grammars”. In: *CoRR* abs/1207.0443 (2012). arXiv: 1207.0443. URL: http://arxiv.org/abs/1207.0443 (cit. p. 136).

[21] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, 2011 (cit. p. 135).

[22] Alexander A. Myltsev. “Parboiled2: A macro-based approach for effective generators of parsing expressions grammars in Scala”. In: *CoRR* (2019). DOI: https://doi.org/10.48550/arXiv.1907.03436 (cit. p. 135).

[23] David S. Wile. “Lessons learned from real DSL experiments”. In: *Sci. Comput. Program.* 51.3 (2004), pp. 265–290 (cit. p. 115).

Plenitude, when too plenitudinous,
was worse than destitution,
for—obviously—what could one do,
if there was nothing one could not?
Stanisław Lem [16]

5 Static Semantics

In Chapter 3, we have discussed how to use generalization, containment, cardinality constraints, and associations to control the set of legal instances of a model. Nevertheless, when working on your own models, you must have arrived at situations when capturing the exact set of desirable instances using a class diagram was either impossible or cumbersome in counter-productive ways. For example, consider the simple class diagram shown in Fig. 5.1. The model captures parent-child relations between people. Every person can have up to two parents, and every person can have some children. The parent-child relation is modeled using two uni-directional references, since Ecore lacks bidirectional associations.

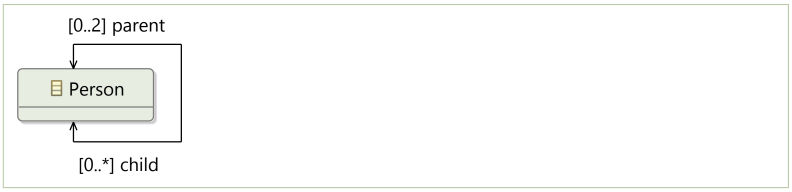


Figure 5.1: *An Ecore class diagram with two unidirectional references (overlayed on top of each other) for the* `Person` *class*

It is natural to require, for any instance conforming to this model, that if person `A` is a parent of `B` then the two persons are distinct, and that `B` is also a child of `A`. Figure 5.2 shows instances violating these two invariants. In the first instance, object `B` (respectively object `C`) is not a child of `A` (respectively `D`). The second and third instances in the figure are variations of circularity problems involving two objects and one object.

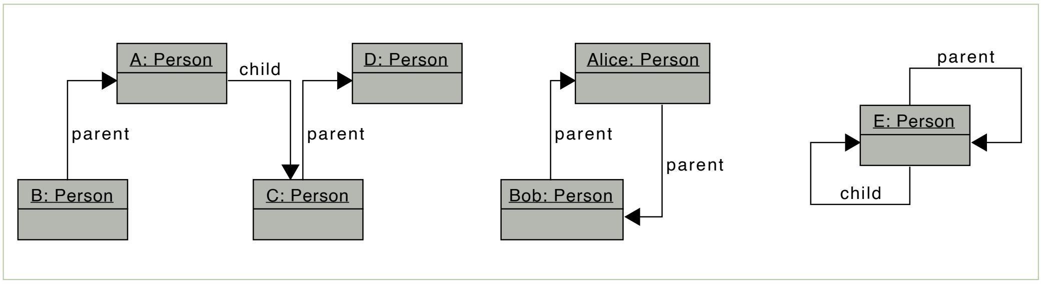


Figure 5.2: *Three undesirable instances admitted by the diagram of Fig. 5.1. The diagrams show complete models, not partial views*

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_5

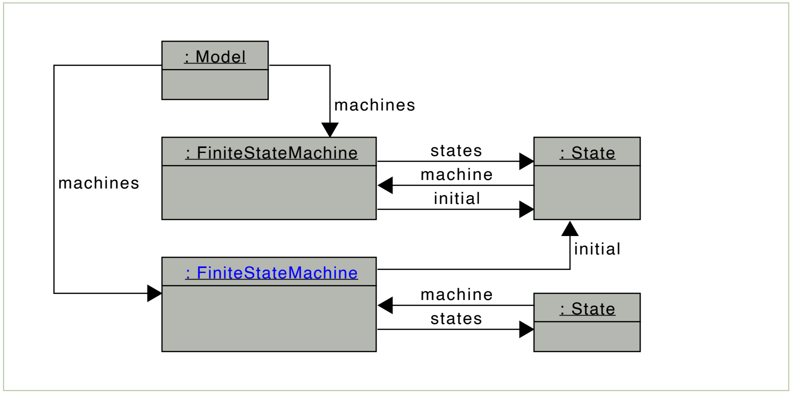


Figure 5.3: *An unexpected instance of the* `fsm` *meta-model of Fig. 3.1, with one machine 'borrowing' its initial state from another one*

In this case, we could fix the class model using a black diamond, to rule out cycles, and a bidirectional association instead of two references,[1] to rule out violations of the inverse of parent–child relations. This would invalidate all the instances of Fig. 5.2. Unfortunately, the problems mount up quickly when we add more intricate constraints. What if we are only interested in instances that contain at least two generations of people? Or families where the two parents hold different passports? The meta-models quickly get large when you start to be precise about all domain constraints. Often it is impossible to capture the desired constraint using just the diagram constructs, due to their limited expressiveness.

In the `fsm` meta-model (Fig. 3.1, p. 53), we would like to require that the initial state of each machine is also its own state. An instance violating this requirement is shown in Fig. 5.3. Working around this problem is cumbersome. We may turn the `initial` reference in Fig. 3.1 into a containment (diamond). This would prevent a state from another state machine from being referenced, at the cost of creating a new problem: the collection of `states` would no longer contain all states, as an object can have at most one owner. For an even more annoying complication, consider the following exercise.

> **Exercise 5.1.** Sketch an instance of a finite-state-machine meta-model, with two machine objects and a transition that crosses between states of the two machines. Is it possible to rule out this instance via meta-modeling?

Problems with capturing constraints precisely are not limited to references and containments in Ecore, which can only express constraints enforcing acyclic hierarchies. They are not limited to object-oriented meta-modeling languages either. We experience the same challenges when building abstract syntax as algebraic data types in functional style. In fact, Ecore has slightly more mechanisms to express constraints than the type systems of mainstream programming languages. As an example, Fig. 5.4 shows an ADT in Scala corresponding to the meta-model of Fig. 5.1. Lines 3–6 define

[1] In Ecore, one can enforce that two references are opposites by setting the `eOpposite` attribute (http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/EReference.html#getEOpposite(), as of 2022/09)

```scala
// This model disallows cycles,
// but also disallows parent-child inversion...
case class Person:
  name: String,
  parent: List[Person],
  child: List[Person]

// The following fails to typecheck
val A: Person = Person ("A", parent = List (B))
val B: Person = Person ("B", child = List (A))
```

Figure 5.4: *A Scala ADT for the Parent/Child example (cf. Fig. 5.1). An attempt to disallow cycles ends up with a model that cannot be instantiated*

```scala
class Person (name: String,
              parent: => LazyList[Person] = empty,
              child: => LazyList[Person] = empty)
```

source: person.scala/src/main/scala/dsldesign/person/scala/Person.scala

Figure 5.5: *An instantiatable pure ADT that can represent inverse parent–child relationships, unlike Fig. 5.4, but cannot prevent cycles*

a class `Person` with collections `parent` and `child`. This class can only be instantiated if parent child relationships are acyclic. In any pure functional programming language, values with cyclic reference structures cannot be created using eager constructors. Unfortunately, this not only disallows cycles but also the duality of parent–child references. Consequently, we cannot represent that an object is a parent of its own child in this design.

One way to work around this is to use side effects and imperative programming: create disconnected objects first, and then wire them up with assignments. This is essentially what Ecore does, and we have seen above that it has its own problems. Alternatively, we can follow the same pattern as in Chapter 3 and model the problem using not object references, but maps and identifiers (cf. Fig. 3.5). There is however no obvious way to statically enforce correctness of such maps, so that one is an inverse of another, or that a transitive closure of map key–value pairs forms no cycles, etc.

Exercise 5.2. Create an ADT representing the abstract syntax of the person example in Scala (or any other functional programming language) that uses explicit person names, and named-based references, like in Fig. 3.5. Specify the 'broken' instances, corresponding to those in Fig. 5.2, that type-check against, and can be constructed successfully with, your ADT.

Figure 5.5 presents an alternative abstract syntax in Scala for our example. In this case, we are using lazy (by-name) references to encode cycles. The `parent` and `child` properties are lazy, they are not evaluated before access. This way we can complete constructing a person object without an immediate access to the parent and child objects. These references need to exist only later, when we ask for parent or child properties in a subsequent computation. We also add default values for the parent/child properties, so that it is easy to construct objects without parents or children.

Figure 5.6 shows example instances of this "lazy" design. All values in this figure type-check and, thanks to laziness, can be explored at runtime without causing stack overflows. The instance (a) demonstrates that we can now represent both parent–child references that are inverses of each other

```scala
// (a) Capturing the parent-child duality (a positive example)
lazy val Mom: Person = Person (name="Mom", child=LazyList (Son))
lazy val Son: Person = Person (name="Son", parent=LazyList (Mom))

// (b) A violation of parent-child duality (a negative example)
lazy val B = Person (name="B", parent=LazyList (A))
lazy val A = Person (name="A", child=LazyList (C))
lazy val C = Person (name="C", parent=LazyList (D))
lazy val D = Person (name="D")

// (c) Circular instances with two objects (a negative example)
lazy val Bob: Person = Person (name="Bob", parent=LazyList(Alice))
lazy val Alice = Person (name="Alice", parent=LazyList (Bob))

// (d) A circular instance with a single object (negative)
lazy val E: Person =
  Person (name="E", parent=LazyList (E), child=LazyList (E))
```
source: person.scala/src/test/scala/dsldesign/person/scala/PersonSpec.scala

Figure 5.6: *Four instances of the ADT in Fig. 5.5. The first instance shows that duality of the parent/child relationships can be represented in a pure manner using laziness. Unfortunately instances (b)–(d) show that all pathological cases of Fig. 5.2 can be represented as well*

like in Ecore. The `Mom` object is the parent of the `Son` object, and the `Son` object is a child of the `Mom` object. This was impossible to represent in the eager design of Fig. 5.4. This relaxation allows modeling of cyclic structures, but it is too weak to control them. We still lack facilities to enforce that some references are inverses of each other while others remain acyclic. Instances (b)–(d) show Scala encodings of the unreasonable Ecore instances from Fig. 5.2. You will experience similar problems, whatever modeling or programming language you are using. The real world invariably calls for more intricate restrictions than modeling languages and type systems are able to express. It is best for us to give up the delusion of a faithful and direct representation of domain constraints in the abstract syntax itself.

Domain-specific models have to adhere to *domain constraints*. The kind of requirements regarding references and cardinalities we discussed above should have been uncovered during domain analysis; see Sect. 3.2, especially question Q4 on p. 50. Repeat the analysis if it fell short. New constraints typically appear *throughout* the language implementation process, even during construction of the back-ends. It would be naive to expect that you can discover all of them in the initial conversation with your users. Expect the set of domain constraints to grow throughout a project.

Example 15. During meetings with subject matter experts, it is both efficient and effective to capture domain constraints in natural language. For instance, the following constraints might have been collected during domain analysis for the finite-state-machine language:

C1 All machines must have distinct names.

C2 All states within the same machine must have distinct names.

C3 For each state machine m, the state designated as the initial state of m is also a member of the collection of states contained in M.

C4 Transitions cannot cross machine boundaries (target and source are in the same state machine).

C5 Each state must be reachable from the initial state in each state machine.

The limitations of the implementation of the modeling language will impose further constraints. For instance, the code generator implemented by the authors for the finite-state-machine language does not handle non-determinism. If you called the generator on a model with non-determinism, it would produce code that fails to compile. Consequently, the example of Fig. 4.5 is not a valid input model for this generator. The non-determinism is present in state `S0` (lines 4–5 both respond to the same input) and in state `S1`. This leads us to formulate an additional constraint, at least until a better generator is developed.

C6 Every two transitions originating in the same state must have different input labels.

Constraints written in English have limited utility. While they are often easier to understand than constraints written in a formal language, they need to be checked and analyzed manually. They are also prone to misinterpretation, as natural language is often ambiguous. To use domain constraints in an automated language-processing tool we need to write them in a machine-processable form, unambiguous, executable, decidable, and testable.

There are two established ways to enforce domain constraints in modeling languages: formal *structural first-order constraints* and *type systems*. We discuss both methods in this chapter, but emphasize structural constraints over type systems. Structural constraints, or constraints for short, are a cheaper and simpler method, suitable for small languages used commonly in model-driven development (Sections 5.1–5.4). In Chapter 6 we present a simple type system and explain when constraints are insufficient and type systems should be used. Of course, the two techniques can be combined in an implementation of a single language to address different problems.

The structural constraints and a type system define the *static semantics* of a language.

Definition 5.1. Static semantics *defines what models are well-formed (valid) by constraining structural connections in the model syntax, so that the model elements are related in a meaningful manner.*

Definition 5.2. *A* well-formed *(valid) model instance is an instance that conforms to the meta-model (it satisfies the diagrammatic constraints)* and *satisfies the domain constraints that have been formulated either using structural constraints or in a type system, or using both means. If a type system is used, then a well-formed instance is also called* well-typed.

Well-formedness should be established right after parsing and the conformance checks performed by the front-end of a tool, the parser. The early enforcement of well-formedness allows other components of a tool chain to be greatly simplified. An explicit definition of static semantics also leads to

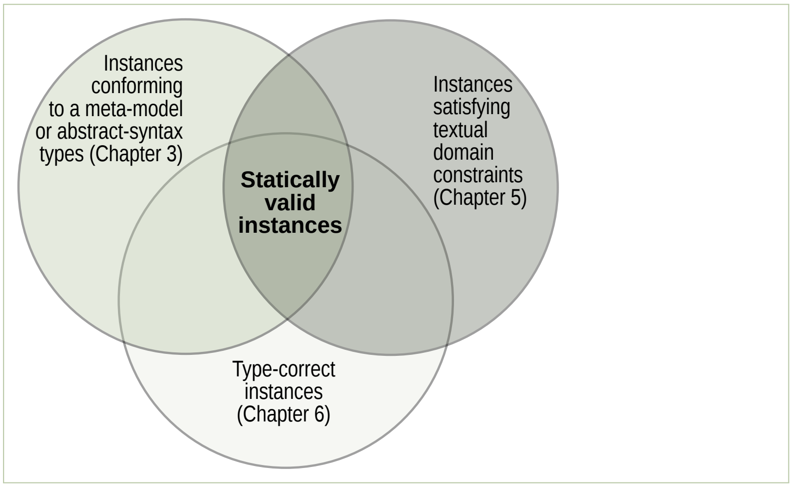

Instances conforming to a meta-model or abstract-syntax types (Chapter 3)
Instances satisfying textual domain constraints (Chapter 5)
Statically valid instances
Type-correct instances (Chapter 6)

Figure 5.7: Meta-modeling and two common methods of defining static semantics for modeling and programming languages. The set of statically valid instances in the center is *the static semantics of the language*

a desirable separation of concerns: the validation and the interpretation of an input are not mixed. It also allows better code reuse in the tool chain, as all tools can reuse the same validator.

5.1 Static Semantics with First-Order Structural Constraints

The easiest way to represent domain constraints is to use logical predicates restricting the connections between model elements. Consider the Constraint **C4** above: *target and source states are in the same state machine*. For a transition object `t`, we can specify **C4** as a Boolean expression in a programming language:

```
t.source.machine == t.target.machine
```

Such executable constraints can be used by tools to automatically validate input models. To program such constraints, a language tool developer needs to be able to reason about restrictions on structures of abstract-syntax trees, to choose the best formulation and the most effective implementation. We devote a few pages to a mathematical interpretation of first-order predicates over models, in order to facilitate development of these reasoning skills.

Definition 5.3. *A* constraint *is a pure (side-effect free) Boolean expression declared over elements of a meta-model, but interpreted over its instances. Its purpose is to restrict the set of valid instances of the meta-model.*

Constraints are declared over meta-model elements, but their semantics impose restrictions on the elements of instances. A constraint's value decides whether an instance is valid or not. In that, constraints resemble meta-models, which also restrict the set of valid instances (cf. Fig. 5.7). If we define the semantics of class diagrams in the same formalism as constraints, we can obtain a unified understanding of the instance space as the intersection of the diagram and the constraints. Tbl. 5.1 defines the core semantics of class diagrams by translation to first-order predicate logic. Once we have interpreted diagrams as first-order formulae, it is straightforward to conjoin more first-order sentences formulating the domain constraints.

Diagrammatic constraints. Let us discuss the formalization in Tbl. 5.1 row by row, just enough to develop a logic-based intuition for reading diagrams. For each class C in the meta-model we introduce a unary predicate of the same name $\mathsf{C}(\cdot)$ that holds precisely for the instance objects x that belong to class C. Here, "unary" means that the predicate has one argument. For the finite-state-machine meta-model of Fig. 3.1 we create predicates Model, FiniteStateMachine, State, Transition, and NamedElement.

If a class D generalizes a class C, we require that the predicate C implies the predicate D: every object of class C is also an object of class D, or, in other words, the set of instances of C is a subset of the set of instances of D. Class D is larger, more general. See the second row in Tbl. 5.1. For our example, this yields the following implications:

$$\forall x.\,\mathsf{Model}(x) \to \mathsf{NamedElement}(x) \tag{5.4}$$

$$\forall x.\,\mathsf{FiniteStateMachine}(x) \to \mathsf{NamedElement}(x) \tag{5.5}$$

$$\forall x.\,\mathsf{State}(x) \to \mathsf{NamedElement}(x) \tag{5.6}$$

There is no corresponding implication for Transition because this class is not generalized by any other class; transitions are not named elements. Since implication is transitive, so is the generalization relation. This cannot be seen in our example, as the hierarchy of generalization is only one level deep. If State had subclasses, they would also be subclasses of

Table 5.1: Mapping core concepts of class diagrams to first-order predicate logic. Notation: $\forall$ = for all (universal quantification), $\to$ = implies, $\wedge$ = and, $|\cdot|$ = the number of elements in a set, $[\mathsf{a};\mathsf{b}]$ = an interval of integers between a and b including both endpoints, $\equiv$ is logical equivalence (equality of logical values). Variables x and y range over objects and values in an instance model. A conforming instance must satisfy all generated constraints simultaneously. For simplicity we assume that names of all references and attributes are globally unique in the meta-model

Class C	$\rightsquigarrow$	A unary predicate $\mathsf{C}(x)$ true iff the type of object x is C
Class D generalizes class C	$\rightsquigarrow$	A constraint $\forall x.\,\mathsf{C}(x) \to \mathsf{D}(x)$
Non-containment reference r from class C to D, $\mathsf{C.r : D}$	$\rightsquigarrow$	A binary predicate $\mathsf{r}(x,y)$ true if reference $\mathtt{r}$ from x points to y and a constraint $\forall x,y.\,\mathsf{r}(x,y) \to \mathsf{C}(x) \wedge \mathsf{D}(y)$
Containment reference from class C to D, $\mathsf{C.r : D}$	$\rightsquigarrow$	Same as the non-containment reference plus a constraint that $\forall x,y.\,\mathsf{r}(x,y) \to \mathsf{owns}(x,y)$, where $\mathsf{owns}(x,y)$ is a special predicated shared between all references in the diagram such that $\forall y.\ \lvert\{x \mid \mathsf{owns}(x,y)\}\rvert \leq 1$.
Cardinality constraint `[a..b]` on reference r in class C	$\rightsquigarrow$	A constraint $\forall x.\mathsf{C}(x) \to \lvert\{y \mid \mathsf{r}(x,y)\}\rvert \in [\mathsf{a};\mathsf{b}]$
Attribute a of type T in a class C, $\mathsf{C.a : T}$	$\rightsquigarrow$	The same as the non-containment reference: a binary predicate $\mathsf{a}(x,y)$ true if the value of a in x is y and a constraint $\forall x,y.\mathsf{a}(x,y) \to \mathsf{C}(x) \wedge \mathsf{T}(y)$, where $\mathsf{T}(y)$ holds iff T is the type of y
References r_1, r_2 are opposite	$\rightsquigarrow$	A constraint $\forall x,y.\,\mathsf{r}_1(x,y) \equiv \mathsf{r}_2(y,x)$

NamedElement. This scheme handles multiple inheritance, too: a class can be generalized by more than one super-class. Its instances are simply instances of all the generalizing classes.

We interpret references as two-argument predicates (binary predicates), as shown in row 3 of Tbl. 5.1. For each reference r we introduce a predicate $\mathsf{r}(\cdot,\cdot)$ relating the referencing and the referenced objects. For instance, for the reference FiniteStateMachine.states in Fig. 3.1 we add a predicate $\mathsf{states}(x,y)$ with the following type restriction:

$$\forall x.\, \forall y.\, \mathsf{states}(x,y) \rightarrow \mathsf{FiniteStateMachine}(x) \wedge \mathsf{State}(y) \tag{5.7}$$

Recall that Ecore only supports uni-directional references, and the machine–states line in Fig. 3.1 is in fact two references, related by a constraint that the two references are dual (opposite). This means that we also have a constraint for the opposite direction (5.8) and a constraint relating the two references (below, cf. the last row in Tbl. 5.1).

$$\forall x.\, \forall y.\, \mathsf{machine}(x,y) \rightarrow \mathsf{State}(x) \wedge \mathsf{FiniteStateMachine}(y) \tag{5.8}$$

$$\forall x.\, \forall y.\, \mathsf{machine}(x,y) \equiv \mathsf{states}(y,x) \tag{5.9}$$

Moreover a machine is composed of states, so it owns them all. This ownership cannot be shared with any other class, as indicated by the black diamond on the reference arrow in Fig. 3.1. Thus, following the fourth row of the table, we require that belonging to a collection of states implies ownership (5.10), and that there is at most one owner for each object in the model:

$$\forall x.\, \forall y.\, \mathsf{states}(x,y) \rightarrow \mathsf{owns}(x,y) \tag{5.10}$$

$$\forall y.\ |\{x \mid \mathsf{owns}(x,y)\}| \leq 1 \tag{5.11}$$

A cardinality constraint limits the number of objects that can be referenced. It can be turned into a restriction on the size of the sets that it defines on each end of a reference. For the states and machine collections of Fig. 3.1 we have the following constraints (cf. row 5 in Tbl. 5.1):

$$\forall x.\, \mathsf{Machine}(x) \rightarrow |\{y \mid \mathsf{states}(x,y)\}| \geq 1 \tag{5.12}$$

$$\forall x.\, \mathsf{State}(x) \rightarrow |\{y \mid \mathsf{machine}(x,y)\}| = 1 \tag{5.13}$$

The first constraint states that if x is a machine, then it has to contain at least one state. The second states that if x is a state, then it has to be owned by precisely one machine.

The set of all constraints describing a diagram fully characterize its set of instances. It is instructive to compare the following definition with the definition Def. 4.19 on p. 96.

Definition 5.14. *Let M be a meta-model, and Φ_M be the* characteristic first-order formula *for M derived using the rules of Tbl. 5.1. The set of all instances (object models) that satisfy the formula Φ_M are the semantics of a meta-model: $[\![M]\!] = \{m \mid \Phi_M(m)\}$.*

Exercise 5.3. Write the characteristic first-order formula (the diagrammatic constraint) for the meta-model in Fig. 5.1. Then consider the two rightmost instances in Fig. 5.2 and convince yourself that they satisfy the constraints. For each constraint ensure that you know which objects are bound to x and y in the quantifiers.

Additional textual domain constraints. Now we can use the logical predicates as a vocabulary to talk formally about instances of a meta-model, to write domain constraints in logic even if they are not expressible diagrammatically. But how do we translate requirement constraints into formal logic? How do we take a constraint, like **C1** *"all machines must have distinct names,"* and make it formal? In short, we bind all mentioned entities using quantifiers and split the body in half using an implication.

While this is not always explicit in English, most constraints take a form of logical implication from a precondition (the antecedent) to a post-condition (the consequent). You can see it in **C1** if it is rewritten to *"all objects that are **machines** must have distinct **names**,"* or to *"if an object is a **machine**, then it has a different **name** from all other **machines**."* We assume a convention here that the preconditions are underlined and the post-conditions follow directly after. Words represented by predicates in the meta-model formalization are bold. This rewrite also makes the binding of machines to quantifiers clearer. We now explicitly use phrases like *all objects*, *an object*, but we are still somewhat loose about names—*has name*—not making it clear that names are instances of a type as well. In a formalization, all entities mentioned in a constraint need to be bound with quantifiers and linked to particular sets representing properties. Consequently, the final formulation of **C1** is even more verbose: "*For all quadruples of objects, where the first two are **machines** and the last two are their **names**, the names must differ.*"

$$\begin{aligned} \forall m_1.\forall m_2.\forall n_1.\forall n_2.\ & \underline{m_1 \neq m_2 \wedge} \\ & \underline{\mathsf{FiniteStateMachine}(m_1) \wedge \mathsf{FiniteStateMachine}(m_2) \wedge} \\ & \underline{\mathsf{name}(m_1, n_1) \wedge \mathsf{name}(m_2, n_2)} \quad \to \quad n_1 \neq n_2 \end{aligned} \tag{5.15}$$

Typically, a meta-model constraint in first-order logic starts with quantifiers naming all the objects involved, followed by a precondition involving types of objects and any structural assumptions about them. The precondition implies the post-condition, so what should hold. The implication is the central structuring element. Recall that an implication holds vacuously if the antecedent is violated. This way the quantifiers range only over values that satisfy the precondition, so over the objects of the correct types that participate in the selected relations.

Another notable pattern visible in **C1** is the inequality condition, $m_1 \neq m_2$, in the precondition. If m_1 and m_2 are equal then their names will also be equal. Indeed, we are interested in the constraint being enforced only for two *different* machines. The word different is, however, typically omitted in English. It is a common mistake of novice constraint writers to forget it also in logical formalizations. Be careful about that!

Recall constraint **C2**: *"all* ***states*** *within the same* ***machine*** *must have distinct* ***names****"*. This is how it reads in the verbose style: *"for all 5-tuples of objects, where one represents a* ***machine****, two represent its two different* ***states****, and two represent* ***their names****, the* ***names*** *must be different in every valid instance."* This is how it looks as a sentence in logic:

$$\forall m.\forall s_1.\forall s_2.\forall n_1.\forall n_2.\\ \underline{s_1 \neq s_2 \wedge \mathsf{states}(m,s_1) \wedge \mathsf{states}(m,s_2)\ \wedge \mathsf{name}(s_1,n_1) \wedge \mathsf{name}(s_2,n_2)}\\ \rightarrow \quad n_1 \neq n_2 \tag{5.16}$$

A careful reader will notice a slight difference between Eq. (5.16) and our encoding of constraint **C1**. The latter does not mention the unary type predicates FiniteStateMachine and State, despite references from the constraint text. These are omitted because the predicate states is unique in the meta-model and it enforces the types of its arguments, cf. Eq. (5.7). We have implicitly used this trick also for the second argument of predicate name, for both **C1** and **C2**—names enforces the second argument to be a name. However, this predicate does not help restrict the first argument's type beyond NamedElement. Since many elements in the model are named, we had to explicitly restrict m_1 and m_2 to be machines in Eq. (5.15). Remember that all our constraints are interpreted *in conjunction* with the diagrammatic constraints. This can save a lot of typing, but, most importantly, it improves the readability of constraints considerably. When we switch from logic to computer languages for writing constraints, many of these redundant type predicates will become implicit navigations. Constraint **C3** demonstrates the benefits of conciseness particularly clearly:

$$\forall m.\forall s.\ \underline{\mathsf{initial}(m,s)} \rightarrow \mathsf{states}(m,s) \tag{5.17}$$

We have used four quantifiers in Eq. (5.15) to introduce four variables. Equation (5.16) had as many as five quantifiers. In logic, a quantifier binds a variable. A correct constraint in logic should have no *free* variables (variables which are not bound). When writing constraints always check whether they contain no free variables—these are invariably a sign of a logical mistake. All variables need to be introduced by quantifiers, otherwise we do not know how to interpret them. Are they arbitrary? Is a single value fixed? Are multiple values possible?

The universal quantification (∀) is much more common in domain constraints than the existential quantification (∃), because we typically enforce domain properties on *all* instances of a type. Often, the universal quantifier is implicit in English; it is implied, or hidden in an indefinite article, especially in formal writing, like requirements documents: *"A state must have a machine it belongs to"* is likely meant to say that *"Every state shall be owned by some machine."* However, existential quantification is also used. It is often used to express lower-bound restrictions that there is at least one entity of some kind or that some sets are not empty. For the sake of an example, let us reformulate Eq. (5.13) using existential quantification. Convince yourself that this and the original formulations are equivalent in our context:

$$\forall x.\ \underline{\mathsf{State}(x)} \quad \rightarrow \quad \exists y.\ \mathsf{Machine}(y) \wedge \mathsf{states}(y,x) \tag{5.18}$$

Recall Constraint **C4** from p. 146: *"Transitions cannot cross machine boundaries. Target and source states must be in the same state machine."* Here is how we can detail this constraint taking the concept of transition as the starting point: *"For any transition with a target state s_2 and a source state s_1 that belongs to machine m, the target state also belongs to m."* Formally:

$$\begin{aligned}&\forall t.\forall s_1.\forall s_2.\forall m.\ \mathsf{source}(t,s_1) \wedge \mathsf{target}(t,s_2) \wedge \mathsf{machine}(s_1,m)\\ &\qquad\qquad \rightarrow \mathsf{machine}(s_2,m)\end{aligned} \tag{5.19}$$

There are many ways to write the same constraint equivalently. We could have started from a model object, not from a transition: *"In every model, and in each machine of that model, if you take a transition belonging to this machine (that is a transition sourced in some state belonging to this machine), its target state also needs to belong to the same machine."* You probably appreciate that the English formulation is much simpler in the original. The formulation in first-order logic is correspondingly more complex as well:

$$\begin{aligned}&\forall M.\forall m.\forall t.\\ &\qquad (\mathsf{machines}(M,m) \wedge \exists s_1.\ \mathsf{states}(m,s_1) \wedge \mathsf{leavingTransitions}(s_1,t))\\ &\qquad\quad \rightarrow (\forall s_2.\ \mathsf{target}(\mathsf{t},\mathsf{s_2}) \rightarrow \mathsf{machines}(m,s_2))\end{aligned} \tag{5.20}$$

Writing constraints, like writing code, is an art and a craft. You either are a genius that can produce optimal formulations instantly, or you must be a craftsman that can predictably refactor proposed constraints towards better formulations. An important goal of this chapter is to help you learn this craft. We can already see above that it helps to (i) wisely choose the starting type (the *context class*), (ii) avoid using more than one implication, and (iii) maintain a simple quantification scheme. The universal quantifiers, all in front of the constraint, are often the simplest form to read, but some constraints require more complex schemes.

These considerations are independent of the concrete programming language used to write constraints. First-order logic is probably the most generic specification language, the basis of most of the languages used in practice. We used it to introduce constraints, to ensure that your intuition is robust with respect to idiosyncrasies of more practical programming and modeling languages. Having said this, the software-oriented specification languages do offer a lot of devices to make your life easier and constraints more readable. Just compare our best bid for **C4**, Eq. (5.19), with the formulation that opened this section:

```
t.source.machine == t.target.machine.
```

While this example lacks a quantifier (just one!), the constraint is clearly made simpler by using navigation instead of predicates and multiple intermediate variables. For this reason, we will switch to using realistic constraint languages in Sect. 5.2.

Exercise 5.4. Recall the determinism Constraint **C6** from Example 15 on p. 146: *For any two transitions sourced in the same state, the input labels must be different.* Formulate this constraint in first-order logic using the predicates following the encoding of Tbl. 5.1.

Even though first-order logic can capture most of the properties we need, it falls short for some specific but important cases. In particular, *connectedness properties* for the model graph, which commonly appear in modeling languages, cannot be captured in first-order logic. One such constraint is **C5** from our example: *"Each state must be reachable from the initial state."* Reachability in a finite-state machine means that the directed graph of transitions must be connected. There must be a directed path from the initial state to any other state.[2] A state that cannot be reached appears useless. Connectedness properties are not limited to contrived mathematical models like finite-state machines. They appear in quite many domain-specific languages that aim to describe processes or physical layout. A stage in a business process that cannot be activated is useless. A room in a building that cannot be reached from the main entrance is likely useless. A track that cannot be entered by a train is useless.

A similar property to connectedness is *acyclicity*. A graph of arrows is acyclic if it is impossible to reach each node from its successors. Thus acyclicity often requires the same expressive power as connectedness. For instance, if you are building an abstract syntax for spreadsheets, each of your instances is a particular sheet containing cells. Typically spreadsheet applications require that there are no dependency cycles between cells. Otherwise calculations cannot be done. If you are modeling Ethernet connections, you may disallow cycles in the node graph. When modeling electric circuits you might want to require them.

Connectivity properties are common, yet first-order logic cannot express them. The problem is that connectedness properties are global properties of a graph, while in first-order logic we can only use predicates that relate a finite number of objects to each other—finite-arity predicates capture only local connections in a graph. To talk about connectedness we need to be able to express *transitive closures* of relations induced by predicates. This cannot be done in classic first-order logic.[3] It requires a second-order logic, where we can constrain not only objects, but also predicates. Below we present a simple formalization of Constraint **C5** that uses a transitive closure, so technically it is written in second-order logic:

$$\forall m.\forall s_1.\forall s_2.\quad \mathsf{initial}(m,s_1)\wedge\mathsf{states}(m,s_2)\quad\rightarrow\quad\mathsf{successor}^*(s_1,s_2) \tag{5.21}$$

where $\mathsf{successor}(s_1,s_2) = \exists t.\mathsf{source}(t,s_1)\wedge\mathsf{target}(t,s_2)$ and $\mathsf{successor}^*$ is the reflexive transitive closure of $\mathsf{successor}$.

[2] The direction of a transition in this case is from its source to its target, as in concrete syntax. This direction for connectedness properties does not necessarily have to be the same as the direction of references in the meta-model (and in the instance), but it very often is the same.

[3] For the interested reader, this follows from the compactness theorem for first-order logics

In verbose English Eq. (5.21) says: *If s_1 is an* ***initial*** *state of* ***machine*** *m, and if s_2 is another of its* ***states****, then s_1 and s_2 should be related by (possibly multiple applications) of the successor relation. A* ***state*** *s_2 is considered a successor of a* ***state*** *s_1 if there exists a* ***transition sourced*** *in s_1 with state s_2 as the* ***target***. Notice that since successor is not defined in our meta-model, so it is not part of our basic vocabulary, we had to define it in addition.

The introduction of a helper predicate successor is merely a convenience. On the other hand, the use of the *reflexive transitive closure* of a predicate, denoted by the asterisk in (5.21) is key. Let us define it semi-formally first:

$$\begin{aligned}\mathsf{successor}^*(s_1,s_n) \quad &\equiv \quad (s_1 = s_n) \ \vee \\ (\exists s_2 \cdots \exists s_{n-1}.\, \mathsf{successor}(s_1,s_2) \wedge &\mathsf{successor}(s_2,s_3) \wedge \cdots \wedge \mathsf{successor}(s_{n-1},s_n))\end{aligned} \tag{5.22}$$

A reflexive transitive closure of a binary predicate is a new binary predicate that is reflexive—so that sucessor$^*(s,s)$ holds, as enforced by the first disjunct—and it holds for direct and indirect successors of the left argument s_1, as enforced by the second disjunct. At the definition time we do not know what are the states $s_2, \ldots, s_{n-1}$. The intermediate states and even their number n are different for each pair of arguments. This is in stark contrast with the fixed arity of predicates in first-order logics. Equation (5.22) cannot be written in first-order logic without the informal dots in the quantification.

More precisely, we define the reflexive transitive closure operator as the smallest predicate successor* satisfying the following equation:

$$\begin{aligned}\mathsf{successor}^*(s_1,s_n) \quad &\equiv \\ (s_1 = s_n) \vee \exists s_2.\, &\mathsf{successor}(s_1,s_2) \wedge \mathsf{successor}^*(s_2,s_n)\end{aligned} \tag{5.23}$$

In this context, the smallest predicate means the predicate satisfied by the smallest number of pairs of states, but still satisfying the equation above. A curious reader will notice that without the minimality requirement, many predicates satisfy Eq. (5.23). In particular, the always-true predicate that holds for any two states satisfies it, too. But the always-true predicate does not capture the connections in the graph at all! It turns out that the smallest solution to Eq. (5.23) is what we want, as it captures just enough states to allow traveling over the successor relation, and not more. It is this definition of a predicate as a minimal solution of an equation that cannot be formalized in first-order logics.

Exercise 5.5. Prove that the predicate successor$^*(s_1,s_n) \equiv \mathsf{true}$ satisfies Eq. (5.23).

A minimum reflexive transitive closure is uniquely defined, and captures all reachable nodes in a graph. Thus if you do not work in logic, but have the power of a programming language at your disposal, the transitive closure operator is naturally replaced by using a depth-first-search (or a breadth-first-search) graph traversal. Therefore, it is important to develop enough intuition to realize when a property needs transitive closure, in order to stop writing quantified sentences and switch to a graph exploration algorithm.

C1. *All machines must have distinct names.*

$$\forall m_1.\forall m_2.\forall n_1.\forall n_2.\ m_1 \neq m_2 \wedge \mathsf{FiniteStateMachine}(m_1) \wedge \mathsf{FiniteStateMachine}(m_2) \wedge \mathsf{name}(m_1,n_1) \wedge \mathsf{name}(m_2,n_2) \rightarrow n_1 \neq n_2$$

```
inv[Model] { M =>
  M.getMachines.asScala.forall { m1 =>
    M.getMachines.asScala.forall { m2 =>
      m1!=m2 implies m1.getName!=m2.getName } } }
```

The quantifications over machines turn into iterations over collection properties. In order to iterate over machines, we shift the context to models (the argument of `inv`). There is no need to bind the `name` objects as we can navigate to the values. We use our own `implies` operator for readability.

C2. *All states within the same machine must have distinct names.*

$$\forall m.\forall s_1.\forall s_2.\forall n_1.\forall n_2.\ s_1 \neq s_2 \wedge \mathsf{states}(m,s_1) \wedge \mathsf{states}(m,s_2) \wedge \mathsf{name}(s_1,n_1) \wedge \mathsf{name}(s_2,n_2) \rightarrow n_1 \neq n_2$$

```
inv[FiniteStateMachine] { m =>
  m.getStates.asScala.forall { s1 =>
    m.getStates.asScala.forall { s2 =>
      s1!=s2 implies s1.getName!=s2.getName } } }
```

The first quantification shifted to the context type in `inv[_]`. Otherwise analogous to **C1**.

C3. *For each state machine m, the state designated as the initial state of m is also a member of the collection of states contained in M.*

$$\forall m.\forall s.\ \mathsf{initial}(m,s) \rightarrow \mathsf{states}(m,s)$$

```
inv[FiniteStateMachine] { m =>
  m.getStates.contains (m.getInitial) }
```

Implications $P(x) \rightarrow Q(x)$ are conveniently turned into membership and inclusion tests ($P \subseteq Q$) if sets of objects characterized by P and Q are available (`contains` in Scala). The implication disappears when the precondition is eliminated (as $\mathsf{true} \rightarrow \phi \equiv \phi$).

C4. *Transitions cannot cross machine boundaries (target and source are in the same state machine).*

$$\forall t.\forall s_1.\forall s_2.\forall m.\ \mathsf{source}(t,s_1) \wedge \mathsf{target}(t,s_2) \wedge \mathsf{machine}(s_1,m) \rightarrow \mathsf{machine}(s_2,m)$$

```
inv[Transition] { t =>
  t.getSource.getMachine == t.getTarget.getMachine }
```

All quantifiers disappear thanks to the use of a suitable context type and navigation.

C5. *Each state must be reachable from the initial state in each state machine.*

$$\forall m.\forall s_1.\forall s_2.\ \mathsf{initial}(m,s_1) \wedge \mathsf{states}(m,s_2) \rightarrow \mathsf{successor}^*(s_1,s_2)$$

where

$$\mathsf{successor}^*(s_1,s_n) \equiv (s_1 = s_n) \vee \exists s_2.\ \mathsf{successor}(s_1,s_2) \wedge \mathsf{successor}^*(s_2,s_n)$$

```
inv[State] {s => reachable (s.getMachine.getInitial,s)}

def reachable (s1: State, s2: State): Boolean =
  BFS (Set(s1), Set()).contains (s2)
def BFS (toSee: Set[State], seen: Set[State]):Set[State]=
  val seen1 = seen union toSee
  if toSee.isEmpty then seen1
  else BFS (toSee.flatMap (succ).diff (seen1), seen1)
def succ (s: State): Seq[State] =
  s.getLeavingTransitions
```

The transitive closure operation is implemented as a custom recursive algorithm (`reachable`), but the main constraint (the first line) is still kept in a simple declarative sentential form.

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/constraints.scala

Table 5.2: Mathematical logic (left) vs implementations of constraints in a programming language (Scala, right). We use our Scala-Ecore integration layer scala/src/main/scala/dsldesign/scala/emf.scala

5.2 Writing Constraints in GPLs

The primary use of static semantics, including first-order constraints, is definitional: to specify the language precisely, to disambiguate what instances of the abstract syntax are valid. If this was the only goal, we could formulate the semantics just in logic, like above. Logic is precise and unambiguous enough. But static semantics also needs to be enforced by tools; a code generator, a simulator, a visualizer, an editor. In fact, any tool that processes models needs to check whether its input is valid. Thus we need a way to *execute* constraints, to check whether they hold for each instance.

Constraints written in a general-purpose programming language are executable by definition. We would like to program the constraints, while maintaining the declarative sentential flavor. A low-level imperative programming style, with multiple functions, loops, and variable updates, would turn what should be a concise sentence into a long story. Based on the analysis of the previous section, we need a fairly high-level language in which we can navigate references, force types of objects, and quantify over sets and types. Finally, we need a way to access models, or to link the syntax of models to a programming language. So far, we used the predicates representing the meta-model to refer to model elements in constraints. In a programming language, the abstract-syntax framework provides these facilities. Instead of predicates, we use types, references, and attributes, exposed by Ecore, MPS, algebraic data types, or whatever other abstract-syntax mechanism you use.

Table 5.2 aggregates the constraints discussed in Sect. 5.1 together with their Scala translations. We developed a small Scala library that makes interaction with the Eclipse Modeling Framework slightly easier and enforces a few conventions.[4] In Tbl. 5.2, constraints are implemented as anonymous functions (lambda expressions) returning a Boolean value. The context object is bound to the sole argument. The function expression is wrapped in a factory call `inv[T]` that represents constraints pertaining to objects of type `T`. The created wrapper object provides simple validation logics that can check whether the constraint holds for all elements of type `T` in a given model.

The omnipresent collection conversions (`asScala`) in Tbl. 5.2 are slightly disturbing. Every time we access an Ecore collection we convert it into Scala. These inessential conversions are caused merely by impedance between the two collection libraries. Ecore uses the Java collection library. We inject the necessary casts to access the modern declarative API of the Scala standard library, including the quantifier functions.

In Constraint **C1**, the four logical quantifiers are replaced by just two collection iterations in Scala, one introducing `m1` and one introducing `m2`. The names of machines are no longer bound using quantifiers. Instead, we navigate to their values: `m1.getName`. The shift from universal quantifiers to collection iteration is deceivingly obvious, not least because the collection iterators use the same names as the logical quantifiers. This shift, however, has significant practical implications. We restrict the logic's ability to

[4] source: scala/src/main/scala/dsldesign/scala/emf.scala

quantify over an entire universe of values satisfying a precondition, to quantifying over values reachable from the context object via navigation. While this limits what we can express, it makes constraint checking decidable. Now the constraints can be executed. If we follow good practices of meta-modeling, all model elements are reachable from the root object anyway. Thanks to the single-partonomy principle (p. 55), the root object can be used as the context for a constraint in the worst case.

Constraint **C1** uses the `implies` operator provided by our library. Implication is a commonly used logical connective in logics, but rarely implemented in programming languages. It is always easily introduced by the logical tautology: $a \to b \;\equiv\; \neg a \vee b$. This is how the operator is injected as an extension method on a Boolean type in our library.

Figure 5.8: A Scala extension adding the implies operator to the `Boolean` *class*

```
extension (a: Boolean)
  def implies (b : => Boolean) = !a || b
```

source: scala/src/main/scala/dsldesign/scala/emf.scala

The Boolean value is implicitly converted to an `ImpliesExtension` object, which provides the new `implies` method. In Scala, methods can be called using infix notation, so the new method works well as an operator. Any language equipped with an extension mechanism will allow a new operator to be added in a similar manner. It is worth the effort, as logical constraints written using implication tend to be more concise, and more readable, than those written using ternary if-then-else expressions. The latter get overly verbose when one of their decision branches becomes constant.

Figure 5.9: An example of an overly verbose if-then-else expression with a constant branch

```
inv[Model] { M =>
  M.getMachines.asScala.forall { m1 =>
    M.getMachines.asScala.forall { m2 =>
      if m1 != m2
        then m1.getName != m2.getName
        else true } } }
```

In some languages (Scala, Python, and JavaScript among them), the comparison operator is extended to Boolean values, enforcing the ordering that false is smaller than true. This ordering coincides with the implication: $a \leq b \equiv a \to b$. Thus the less-than operator may provide a cheap way to formulate implication: `a <= b`. Unfortunately, the ASCII symbol for less-than resembles the implication arrow in the opposite direction. It is fairly easy to misread the above as `b implies a`, while it really means `a implies b`. Thus we find it hard to recommend this practice, unless consistently enforced by all developers involved in a project.

Sometimes, we can eliminate an implication entirely. In Constraint **C3**, the implication between predicates has been replaced by a set-membership test. In general, an implication between predicates can be replaced by inclusion of sets they characterize, if these are accessible through navigation, or through the available API. As soon as the precondition has been entirely captured by other means, the implication from a precondition to the post-condition can be removed. This happens if we express the precondition in other ways, for instance filtering by types or navigation (also in **C4**).

Exercise 5.6. Implement Constraint **C6** in a programming language of your choice.

Constraint **C4** is a very simple example of what is often called a *commutativity constraint*. Two ways to navigate from the context object to some target objects should be consistent. This is much easier to see in a GPL than in the logical formulation, because of explicit navigation. If a transition does not cross machine boundaries then navigating from a transition to a machine results in the same machine object whether via the source or via the target link. Many meta-models contain cycles that should be commutative when navigating. Systematically inspecting cycles in the meta-model to identify commutativity (sometimes called diagram chasing) is an established way to identify validation constraints for instances. Indeed, a careful reader will notice that commutativity constraints are a generalization of `EOpposite` duality in Ecore diagrams (Why?).

Constraint **C5** is a special case in our table. Recall from Sect. 5.1 that this constraint is not first-order. It needs to compute a transitive closure, implemented here using a breadth-first search. The recursion breaks the sentential style of the constraint. We wrap the computation into a Boolean function `reachable` to nevertheless be able to state the main constraint declaratively. It is useful to separate sentential constraints from computationally heavy aspects in such cases, implementing helper functions that serve as *derived attributes* of objects. Think of `reachable` and `succ` as if these were new properties of `State` that might be reused in other hypothetical constraints that themselves can be written in a sentential form. If we add a type system or type inference mechanism to a DSL (Chapter 6) we can also integrate the inferred type of an object, or whether an object type-checks against a given type annotation, as a derived property into declarative first-order constraints.

Exercise 5.7. The implementation of Constraint **C5** in Tbl. 5.2 is potentially very inefficient. The reachable state space of the automaton is computed from scratch every time a constraint is checked. Discuss how to redesign this implementation to only compute the reachable state space once per machine, which should save computation time if the constraint is checked on many states.

Everything we said above applies not only to validating abstract syntax in Ecore, but to any abstract syntax that is exposed to a programming language as values and types. This includes abstract-syntax ADTs in functional programming. The only thing that changes is that you are using different types and functions to express the constraint. Consider the following exercise.

Exercise 5.8. Implement the Constraints **C1**–**C6** for the abstract syntax presented in Fig. 3.5 on p. 60. The abstract-syntax ADT code is available at fsm.scala/src/main/scala/dsldesign/fsm/scala/adt.scala

Scala: Constraint C2 repeated from Tbl. 5.2. We use `asScala` to convert from Java collections used in the EMF API. The `implies` and `inv` functions are implemented in the book's library.

fsm.scala/src/main/scala/dsldesign/fsm/scala/constraints.scala

```
val C2 = inv[FiniteStateMachine] { m =>
  m.getStates.asScala.forall { s1 =>
  m.getStates.asScala.forall { s2 =>
    s1!=s2 implies s1.getName!=s2.getName } } }
```

Python: Very concise thanks to the dedicated comprehension/query syntax. The quantifiers come first, a precondition at the end. Type checking only at runtime. PyEcore helps to use DSLs in robotics and data science projects.

fsm.py/constraints.py with pyecore

```
C2 = lambda m: all ( s1.name != s2.name
  for s1 in m.states for s2 in m.states if s1 != s2)
```

JavaScript: No type checking, not even at runtime; C2 might hold on any object that has 'states' and 'name.' We cast lists to `array` as the standard list API is too weak. Note the quirky use of the less-than operator as implication, in the "wrong" direction. Ecore.js helps development of DSLs for the web, server- and browser-side.

fsm.js/constraints.js with ecore.js

```
var C2 = m =>
  m.get('states').array().every ( s1 =>
  m.get('states').array().every ( s2 =>
    (s1!=s2) <= (s1.get('name')!=s2.get('name')) ))
```

Java: the most verbose of the shown languages; with a bit underdeveloped collection API. We cast lists to streams, in order to access quantifier functions. The constraint could be made more terse using a functional-programming library.

fsm.java/src/main/java/dsldesign/fsm/java/Constraints.java

```
Function<FiniteStateMachine, Boolean> C2 = m ->
 m.getStates().stream().allMatch ( s1 ->
 m.getStates().stream().allMatch ( s2 ->
  s1==s2||!Objects.equals(s1.getName(),s2.getName())));
```

Groovy and **Kotlin** conveniently extend Java collections (using extension methods) with higher-order functions. The default argument "`it`" in anonymous functions simplifies the constraints slightly. Both examples access the Java API generated by EMF. Kotlin is interesting if your DSL is to operate on Android devices.

fsm.groovy/src/main/groovy/dsldesign/fsm/groovy/Constraints.groovy

```
def C2 = {
 it.states.every { s1 ->
 it.states.every { s2 -> s1==s2 || s1.name!=s2.name }}}
```

fsm.kt/src/main/kotlin/dsldesign/fsm/kotlin/Constraints.kt

```
val C2: (FiniteStateMachine) -> Boolean = {
  it.states.all { s1 ->
  it.states.all { s2 -> s1==s2 || s1.name!=s2.name }}}
```

Xtend makes the "`it`" argument even more expressive, opening its namespace like Java does for `this`. You can't even see "`it`" in the example, where `states` really means `it.states`.

fsm.xtend/src/main/xtend/dsldesign/fsm/xtend/Constraints.xtend

```
val (FiniteStateMachine) => Boolean C2 = [
  states.forall [ s1 |
  states.forall [ s2 | s1==s2 || s1.name!=s2.name ]]]
```

C#: A Java-like shape of C2 is possible in C#, but we show LINQ syntax to demonstrate a different style, aiming at programmers experienced with database queries.

fsm.cs/Program.cs with .NETModelingFramework

```
Func<IFiniteStateMachine,bool> C2 = m => (
  from s1 in m.States from s2 in m.States
  where s1!=s2 select s1.Name==s2.Name).All (x => !x);
```

F#: We show both the LINQ (first) and the functional (second) form for C2. Note that the F# LINQ interface includes a universal quantifier, which makes C2 less cryptic than in C#. The functional formulation suffers from type-impedance between collection libraries (`Seq.toList`), like many other languages.

fsm.fs/Program.fs with .NETModelingFramework

```
let C2: IFiniteStateMachine -> bool = fun m -> query {
  for s1 in m.States do for s2 in m.States do
    where (s1 <> s2) all (s1.Name <> s2.Name) }
let C2a: IFiniteStateMachine -> bool = fun m ->
  m.States |> Seq.toList |> List.forall (fun s1 ->
  m.States |> Seq.toList |> List.forall (fun s2 ->
    s1 = s2 || s1.Name <> s2.Name) )
```

Table 5.3: Constraint C2 from Tbl. 5.2 implemented in nine programming languages for comparison

Obviously, Scala is not the only language in which one can write validity constraints. Most modern general-purpose programming languages are perfectly suitable for the task. To illustrate the point, we formulated Constraint **C2** in nine mainstream languages. Table 5.3 explores various presentation styles, while maintaining the same computational intention. We do not seek a smarter or a more idiomatic formulation. We display differences between languages, not between different ways to write a constraint. And the differences turn out to be minor. Consequently, we recommend that, in typical projects, where a DSL implementation is just a task in a larger system endeavor, you write static semantics constraints in the language determined by other system requirements. This is likely going to decrease the maintenance cost, ensuring that developers familiar with the implementation language are available to evolve the DSL. Using a specialized constraint language makes sense if you are developing many languages (for example a language-engineering consultant). Otherwise, the investment is probably not justifiable.

The book code repository contains not only the source code of all the nine constraints from Tbl. 5.3, but also the driver code that initializes the relevant Ecore library, loads the model, and executes the test of the constraint on several instances. You can use these examples to scaffold your own projects interacting with Eclipse EMF, in any of the nine programming languages. That we can write this example in nine different programming languages is a testimony to how recognized Ecore is as a technology. All the nine programs use the same finite-state-machine meta-model, and the same test instances stored in the XMI format. This also means that you can use XMI and Ecore as an interchange platform for language-oriented data. You can write and reuse language tools implemented in various programming languages, and protect yourself from being captured by a single vendor.

In the table, all the examples in JVM languages (Scala, Java, Groovy, Kotlin, and Xtend) use the code generated by the main implementation of Ecore from the Eclipse Modeling Framework. EMF generates an implementation of a meta-model as Java classes and interfaces. Any JVM language can interact with these classes. The only problem, as seen for Scala, might be inefficiencies related to differences in the standard libraries. In Tbl. 5.3, Scala and Java (sic!) use conversions between legacy collections and other types. In Scala, as mentioned above, the standard library provides a suitable higher-order API. In Java, the standard lists lack such an API, so it is useful to convert lists to streams, which have a more modern functional interface. Groovy, Kotlin, and Xtend all provide extension methods that enrich Java APIs suitably, so the code is less cluttered.

The Python implementation is based on the pyecore library.[5] The JavaScript implementation is based on the ecore.js library.[6] What is inter-

[5] https://github.com/pyecore/pyecore, retrieved 2022/09

[6] https://github.com/emfjson/ecore.js, at the time of writing the implementation does not enforce `EOpposite` constraints between references. You may need to maintain or check them yourself.

esting about both of these implementations is that no code is generated. The interpretation of meta-models happens entirely at runtime, which can be conveniently done in dynamic languages. The C# and F# examples use the .NET Modeling Framework (NMF),[7] which technically is not a reimplementation of Ecore, but a similar independent modeling framework that can import Ecore meta-models. It uses the same XMI format for instances as EMF.

Importantly, whatever programming language we use the constraints remain written in pure declarative style. The structure of the model and the values of properties are not modified during validation—a standard contract between the validator and other components in the tool chain. This contract is important, as the validation logic might be executed multiple times, not always under your control. For instance, if the validator is integrated into an Eclipse editor, constraints are executed every time a file is saved, sometimes every keystroke a user types. Obviously, a user creating a model should not see her model changed by the validation logic while typing.

In all nine languages, we have used anonymous functions to represent constraints, with the context element bound to the sole argument. Several languages (Groovy, Kotlin, and Xtend) provide a special variable named '`it`' that serves as an implicit formal argument to a lambda expression, and makes writing constraints slightly more concise. In Scala, the underscore can be used similarly, but it only works well if you have to refer to the context object once. (This is why we do not use it.) Furthermore, several languages support properties for objects that allow for set/get prefixes to be dropped from access methods; hereunder Python, Groovy, Kotlin, Xtend, C#, and F#. Technically, Scala also supports such syntax, but this would require an extension to the code generated by Ecore for Java, whereas Groovy, Kotlin, and Xtend achieve this without any additional code, as their attribute implementation is based on conventions.

More interestingly, Python, C#, and F# provide a query-like syntax, which brings the first-order constraints to resemble database queries. When a first-order property is written as a query, three components are distinguishable: (i) a *binding* of an iterator variable name to a set (`for/from/for`), combined with (ii) a filter expression that serves as a precondition (`where/select/if`), and (iii) a quantifier to establish the result (`all/All/all`). The relational encoding exploits a classic result from database theory due to Codd [4] that relational queries and first-order predicates over data elements are equally expressive and sufficiently rich to specify many practical data restrictions. Programmers with extensive database experience may find it easier to read and write constraints that resemble database queries.

Equality tests are common in constraints, in both pre- and post-conditions, so you need a very good understanding of the semantics of equality in the used programming language. Equality testing with complex objects and null values easily gets subtle. For example in Java, a test `a.equals (b)` can only be made if `a` is not null, thus you need to test for `a == null` separately.

[7] https://github.com/NMFCode/NMF, retrieved 2022/09

On the other hand, the test `a == b` might be misleading. For instance, two identical String objects are not equal in Java if they are not physically at the same memory location. For exactly this reason we are using a helper function `Object.equals` in our example for Java. Consider how your language executes equality on complex objects that might potentially be null. Remember to test these cases, to rule out possible misconceptions. A mistake here easily flips a constraint value between true and false.

Exercise 5.9. Implement all constraints from Tbl. 5.2 in your favorite programming language. Test Ecore instances can be found under fsm/test-files/ in the book code repository. Alternatively, create your own definition of abstract syntax and program against it. How well does the abstract-syntax model support establishing constraints? Are there any design issues with it? How well does your programming language support writing constraints? Consider the size and readability of your constraints against the examples in Tables 5.2 and 5.3.

Exercise 5.10. The Constraint **C2** could be equivalently formulated in English as follows: "*In every finite-state machine, the set of states of this machine has to be the same cardinality (size) as the set of names of these states.*" This basically means that there are no duplicate state names. Implement **C2** using this formulation in your language of choice. Discuss the difference in computational complexity of the original and the new formulation. Which of the two formulations is more readable in your opinion? Why?

5.3 Specialized Constraint Languages for Modeling

General-purpose languages are hard to beat when it comes to ease of integration with the rest of your project, and the accessibility and familiarity of the basic tooling. Install an interpreter or a compiler—and you are ready to go! Almost no setup and no configuration pains. However, sometimes it is practical to integrate constraints with a meta-model, not with the processing tools. This is where specialized constraint languages and tools can help. Using specialized languages may also help to provide more functionality than just evaluation of constraints—the only functionality that programming languages provide. For instance, we can use automatic instance generation to create ad hoc instances for testing tools.

Object Constraint Language. When meta-modeling in Ecore or UML, the *Object Constraint Language* (OCL) is a natural choice of a specialized constraint language, as it is particularly well integrated with these host languages. OCL is a formal language originally designed to write expressions associated with UML models, including well-formedness constraints and other formal specifications. It has been later integrated with Ecore and with several other languages including model-transformation languages such as QVT and ATL discussed in later chapters. OCL is a strongly typed declarative language, based on first-order predicate logic with a programmer-friendly syntax. For example, quantifiers are disguised as collection operations, and native navigation in object graphs is provided, so that we do not have to write awkward navigations using predicates like in Sect. 5.1. OCL

```
package fsm : _'dsldesign.fsm' = 'http://www.dsldesign.org/dsldesign.fsm' {
  abstract class NamedElement {
    attribute name: String[1];
  }

  class Model extends NamedElement {
    property machines: FiniteStateMachine[*|1] { ordered composes };
    invariant C1: machines->forAll (m1, m2 | m1 <> m2 implies m1.name <> m2.name);
  }

  class FiniteStateMachine extends NamedElement {
    property states#machine : State[+|1] { ordered composes };
    property initial : State[1];
    invariant C2: states->forAll (s1,s2 | s1 <> s2 implies s1.name <> s2.name);
    invariant C3: states->includes (initial);
    invariant C5:
      let reachable: Set(State) =
        initial->closure (s | s.leavingTransitions->collect (t: Transition|t.target))
      in states->forAll (s | reachable->includes(s));
  }

  class Transition {
    property target: State[1];
    property source#leavingTransitions : State[1];
    attribute input: String[1];
    attribute output: String[?];
    invariant C4: source.machine = target.machine;
  }

  class State extends NamedElement {
    property leavingTransitions#source : Transition[*|1] { ordered composes };
    property machine#states : FiniteStateMachine[1];
  }
}
```

Figure 5.10: *Constraints C1-C5 written in OCL, embedded in a textual representation of the meta-model of Fig. 3.1*

allows new functions to be defined, including recursive functions. It is, thus, more expressive than first-order logic. OCL can express transitive closure.

OCL specifications are meant to define invariant conditions that must hold for the system being modeled or queries over objects described in a model. Each OCL expression is related to an instance of a model element, called a *context*, in line with how we used the term above. The keyword `self` returns a reference to the context object. OCL expressions are pure, i.e., their evaluation cannot alter the instance over which they are evaluated. Thus OCL is a very good match for our definition of static semantics constraints (cf. Def. 5.3, p. 148).

Figure 5.10 presents the running example using OCL. More precisely, it presents *both* the finite-state-machine meta-model *and* the associated constraints using the Eclipse *OCLinEcore* textual syntax. The figure shows the same model, the very same file, as in Fig. 3.1 on p. 53, but opened in a different editor, using a different textual concrete syntax, not the graphical syntax of Fig. 3.1. Note the same concrete and abstract classes, the same generalization hierarchy, the same properties, and the same cardinality constraints. The textual syntax allows us to conveniently show OCL constraints inline. Each constraint is introduced with the keyword `invariant` and a label.

Constraint **C1** is found in Line 8. Observe two interesting features: a binary universal quantifier `forAll` that iterates over pairs of objects, and the built-in implication operator. Both of these extensions contribute positively to the simplicity of the formulation. The same advantages are observed in Line 14 for **C2**. Compare how this constraint has been written in general-purpose programming languages in Tbl. 5.3.

The OCL standard library provides a rich set of logical and set operations, and the usual operations for collection manipulation. All Ecore meta-model types are directly accessible. We gather the operations used in our examples along with a few extras in Tbl. 5.4. The `closure` operation deserves a longer discussion. We used `closure` in the implementation of **C5** in Line 18. It computes the reflexive transitive closure of a binary relation provided as a lambda expression. We run closure on a collection of elements of some type `T`. As an argument, we provide a function f that given an element of type `T` computes a new collection containing more elements of the same type. The closure computation will obtain a new collection by applying f to each element in the input, and then union the result with the input. This will be repeated until a fixed point is reached, so until applying the function f no longer gives any new elements. We encourage the reader to compare this definition, and the formulation of the constraint, to our discussion of computing transitive closure to find the set of reachable states in Sect. 5.1.

Exercise 5.11. Implement Constraint **C6** from p. 147 in OCL.

Exercise 5.12. The binary universal quantifier in Line 8 (Fig. 5.10) is quite convenient for writing constraints relating multiple elements of the same type. Most programming languages only provide unary quantifiers in standard libraries, but it is rather straightforward to implement more quantifiers on your own. Implement binary and ternary universal and existential quantifiers in a programming language of your choice. The book source code provides Scala implementations in scala/src/main/scala/dsldesign/scala/emf.scala as an example. Reimplement constraints C1 and C2 using the new quantifiers.

Often when writing constraints, you discover that the meta-model designers have not included object properties that would be useful when programming against the model. If these properties are derivable from the other existing properties in the model, we can use a `let` expression to introduce a function computing the new derived attribute. We did exactly this in lines 17–19 in Fig. 5.10. Unfortunately, a `let` expression introduces a new name only in the scope of a single constraint (in our case **C5**). What if the new attribute should be accessible from many places? Also from other constraints?

OCL supports *derived properties* (derived attributes), to address this use case. Derived attributes are injected in all instance models and computed when needed. Figure 5.11 shows an example. In the present meta-model, the initial state is an attribute of a state-machine object (`initial`). There is

`and or xor not implies`	The essential Boolean connectives.

`let f (x: T1): T2 = ...`
`in ... e ... end`
	Introduce a new function `f` (or a new value) accessible in expression `e`.

`if ... then ... else ... endif`	Ternary conditional expression (if-then-else expression). Corresponds to “`... ? ... : ...`” in C-like languages.
`ss->includes (s)`	True iff the collection `ss` contains element `s`.
`ss->includesAll (tt)`	True iff the collection `ss` contains all elements from the collection `tt`.
`ss->isEmpty (); ss->notEmpty ()`	True iff the collection `ss` is empty (respectively not empty).
`ss->size ()`	Return the number of elements in collection `ss`.
`ss->intersection (tt)`	A collection of elements shared by collections `ss` and `tt`.
`ss->including (s)`	A collection containing the element `s` and all elements of collection `ss`.

`ss->forAll (s1, ..., sn |`
`f (s1, ..., sn))`
	True iff `f` holds for all selections of `n`-element tuples from a collection `ss`.

`ss->exists (s1, ..., sn |`
`f (s1, ..., sn))`
	True iff `f` holds for at least one `n`-element tuple from a collection `ss`.

`ss->select (s | f (s))`	Filter `ss` so that it contains only elements on which `f` is true. This function is known as ‘filter’ in some other languages.
`ss->collect (e | f (e))`	Computes a collection of elements derived from `ss` using `f`. The function `f` returns a collection itself. In other languages this is known as `flatMap` or a bind.
`ss->closure (s | f (s))`	Compute the reflexive transitive closure of `f` by applying it repeatedly (starting with `ss`) until a fixed point is reached.

`ss->iterate (e, z = ini |`
`f (e, z))`
	Iterate `f` over `ss`, where `z` is the current state of the iteration (initially `ini`), and `e` binds to consecutive elements. The function `f` computes a new value of `z`. The last one is returned. Known as ‘fold’ or ‘reduce’ in other languages.

`ss->isUnique (s | f(s))`	Holds iff `f` evaluates to a different value for each element in the source collection `ss`.
`T.allInstances ()`	A collection of all instances of a given type `T` (discouraged; better use context, or navigate to the right subset).
`s.oclIsTypeOf (T)`	True iff the actual type of `s` is `T` (ignores generalization).
`s.oclIsKindOf (T)`	True iff the type of `s` is `T` or its sub-type (observes generalization).

Table 5.4: An abridged reference list of OCL operations and expressions. The argument in lambda expressions can be omitted. It defaults to `self`*. A complete reference of the Eclipse implementation of OCL is available at http://help.eclipse.org/oxygen/topic/org.eclipse.ocl.doc/help/GettingStarted.html, as of 2022/09*

```
1 class State extends NamedElement {
2   ...
3   property machine#states : FiniteStateMachine[1];
4   property isInitial: Boolean [1] { derived, volatile }
5   { derivation: self.machine.initial = self; }
6 }
```

Figure 5.11: *An example of a new derived attribute* `isInitial` *added to the* `State` *class using OCL*

no easy way to check for a given state whether it is an initial one. One needs to navigate upwards to the containing machine object, and compare the self reference with its initial state. This operation can be automated and linked to a derived attribute (Lines 4–5 in the figure). Now we can simply check `s.isInitial` on any state `s`. OCL's derived properties resemble extension methods from general-purpose programming languages. Exercise 5.24 explores this connection.

Exercise 5.13. Rewrite Constraint **C3** in the context of `State` class using the `isInitial` attribute.

There are several independent implementations of OCL. To write this chapter we used the so-called *Pivot OCL* implementation from the Eclipse MDT project.[8] However, the differences between OCL dialects are relatively minor, and you should use the variant that integrates best with the rest of your tool chain. There are also derivative languages that offer added functionality and usability. For example, EVL is a validation language with very similar constraint syntax to OCL.[9] It adds support for modeling dependencies between constraints (e.g., if a constraint fails, another one should be ignored), customizable error messages, and inter-model constraints.

OCL provides the ability to attach constraints to models, instead of committing to a particular programming language. A standard specification [17] and several implementations define how constraints are *evaluated*. Fundamentally though, this is comparable to what any general-purpose programming language offers for defining static semantics, as we have seen in Sect. 5.2. Alloy, and a few related languages, add other interesting abilities: to check whether a set of constraints is *consistent*, to check what properties they *entail*, and to generate *instances* of various shapes. These in turn can be employed to automatically create test cases, or even synthesize programs.

Alloy. Alloy [12] is a textual structural-modeling language that corresponds (roughly!) to class diagrams combined with OCL constraints, but unified in a single syntax. The first user experience resembles the OCL-in-Ecore editor (whose syntax was used in Fig. 5.10). Alloy's semantics is, however, more restricted than OCL. Arbitrary recursion is not allowed, and any execution is of finite bounded size. It does support a transitive closure though. This allows Alloy tools to provide all the additional computational support.

Figure 5.12 presents the finite-state-machine example in Alloy. Alloy is a relational modeling language. The main building blocks are signa-

[8] https://wiki.eclipse.org/OCL/Pivot_Model, seen 2022/09
[9] https://www.eclipse.org/epsilon/doc/evl/, seen 2022/09

tures defining sets of objects that can enter in relations with other objects. Signatures resemble classes in object-oriented languages. More precisely, they model single-column database relations and are close relatives to unary predicates used for types in Sect. 5.1. We have seven signatures in Fig. 5.12: `Name`, `Label`, `NamedElement`, `Model`, `FiniteStateMachine`, `State`, and `Transition`. The first two, `Name` and `Label`, introduce names and labels as types. Alloy supports limited modeling with strings. For our purposes, it is more practical to create two sets (names and labels) that have no further structure and no contents except their own identity, and use them to label named elements and transitions. This will allow us to write constraints about uniqueness of names and labels without concern for the character contents of strings. This will also help Alloy tools to work more efficiently with our model.

In Line 4, we introduce a signature for named elements. Similarly to Ecore and Java, we mark it "abstract," telling Alloy to never directly instantiate it, but to instantiate only its specializations. A named element has a single attribute of type `Name`. The keyword `one` in Line 4 means that there is exactly one name assigned to every instance of `NamedElement`, corresponding to a cardinality constraint `1` or `1..1` in class modeling. It is important to understand that even though `name` is written syntactically as if this was an attribute contained in `NamedElement` it really denotes a binary relation, a subset of the Cartesian product `NamedElement`×`Name`. It relates named elements to their names. Later, when we write constraints in Alloy, we can use attribute names as first-class binary relations (sets of pairs), which allows to use set theory algebra to write constraints, making them very compact and easier to handle for tools.

In Line 6, the document root is declared, as a singleton signature (`one`). A singleton, referring to the singleton pattern [9], means that Alloy tools will always create exactly one instance of the `Model`, as we intended for our use case—so far we always considered one model at a time. The signature has an attribute `machines` relating the single `Model` instance to at least one `FiniteStateMachine` instance. The `some` keyword on the `FiniteStateMachine` type is a cardinality constraint. It means "one or more" (`1..*`). Observe that `Model` extends `NamedElement`, so it also has the attribute `name` inherited from `NamedElement`. The other signature definitions follow analogously. The keyword `set` (lines 10 and 14) means "any number," the same as "`*`", "`n`", or "`-1`" in class modeling. The cardinality `one` means "exactly one" (`1..1`) and `lone` means "at most one" (`0..1`).

Being a relational language, Alloy has no first-class support for containment and partonomy constraints. Nesting of properties does not guarantee that containment is enforced. Intuitively, all Alloy properties are like references in UML without the black diamond. The signatures in our example allow for the same state to be shared by two state machines. This happens by relating both to the same state instance in the `states` relation. Figure 5.13 shows an instance generated for the model consisting of the

```
 1 sig Name { }
 2 sig Label { }

 4 abstract sig NamedElement { name: one Name }

 6 one sig Model extends NamedElement {
 7   machines: some FiniteStateMachine }

 9 sig FiniteStateMachine extends NamedElement {
10    states : set State,
11    initial: one State }

13 sig State extends NamedElement {
14   leavingTransitions: set Transition,
15   machine           : one FiniteStateMachine }

17 sig Transition { target: one State,
18                  input : one Label,
19                  output: lone Label,
20                  source: one State }

22 fact { Model.machines = FiniteStateMachine }
23 fact { FiniteStateMachine.states = State }
24 fact { State.leavingTransitions = Transition }
25 fact { machine = ~states }
26 fact { source = ~leavingTransitions }

28 fact C1 { #FiniteStateMachine.name = #FiniteStateMachine }
29 fact C2 { all m: FiniteStateMachine |
30           #m.states.name = #m.states }
31 fact C3 { initial in states }
32 fact C4 { source.machine = target.machine }
33 fact C5 { all m: FiniteStateMachine |
34           m.states in m.initial.*(leavingTransitions.target) }
```

source: fsm.als/fsm.als

Figure 5.12: *The* `fsm` *meta-model in Alloy, with explicit partonomy constraints and the initial state constraint*

first twenty lines of Fig. 5.12. Notice that a single state is shared by two machines in the instance. Similarly, because cardinality is only restricted on the far end of references, it is possible to create instances of machines, states, and transitions that are not contained in any other objects.

In lines 22–26, we establish the containment constraints explicitly. We first require that that all finite-state machines are related to a model, that all states are related to a finite-state machine, and all transitions are leaving some state. These together disallow objects that float freely, outside a partonomy. To understand the syntax of these constraints, note that `Model.machines` computes the (database-like) join of the set of models with the `machines` relation, ultimately resulting in the set of all machines that are related to the model. The constraint requires that this set is the same as the set of all finite-state machines (l. 22). The constraints in lines 23–24 follow the same pattern. To restore the requirements that objects are not shared by more than one container in a partonomy, we enforce the duality of navigation (l. 25–26). The first constraint says that the `machine` relation is the

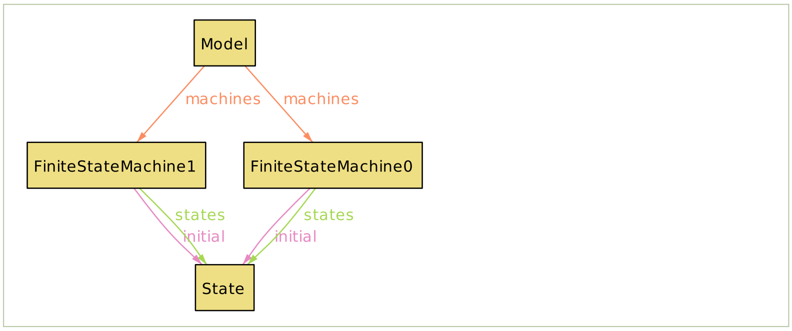


Figure 5.13: An instance of an `fsm` with two machines sharing a single state. This instance has been generated by the Alloy analyzer for the first twenty lines of the model in Fig. 5.12

inverse of the `states` relation. The second imposes that the `source` relation is the opposite of the `leavingTransitions` relation. This not only enforces synchronization of references in the style of Ecore's `EOpposite`, but also disallows sharing. Consider the example: since each state can point to exactly one `machine`, its inverse, `states`, cannot link the same state to multiple machines. It is easy to build a similar argument for transition sources. There is no corresponding constraint for containment of machines in a model—this one is not needed, because the model signature defines a singleton.

In lines 1–26 we have represented the `fsm` meta-model in Alloy. Now we can, finally(!), write the static semantics constraints. Constraint **C1** is shown in Line 28. The formulation uses set cardinality. The statement that all machines have unique names is equivalent to the set of machines and the set of their names being equal size. In Alloy the operator `#x` returns the size of the set `x`. Constraint **C2** is implemented in the same manner, just restricted to a subset of states pertaining to a single machine, using a quantifier.

In Line 31, we require that the initial state is an own state of a machine (**C3**). More precisely, `initial` is a relation (a set of pairs) linking each machine to exactly one state, and `states` is also a relation with a higher cardinality of the image. The constraint states that the former (seen as a set of tuples) is a subset of the latter: if a machine is related to a state in the `initial` relation, then it is also related in the `states` relation.

Constraint **C4** (transitions cannot cross machine boundaries) is implemented using the same principle. We require that the relations created by joining the transition objects with machine objects via source states and via target states are the same. Both relations pair transitions and machines in tuples (t_i, m_j). Because there is only one entry in each set for each transition t_i (why?), the equality of the relations entails that for each single transition its source and target must be the same (m_j). If these two relations differed, there would be at least one transition which would be paired with a different machine via the source than via the target state. Constraints **C1**–**C4** demonstrate that when working with relations like with sets, we can often drop quantifiers. In relational languages, this is an additional instrument for making constraints concise (on top of choosing the right context type).

Finally, in the reachability Constraint **C5** we use Alloy's transitive closure operator (l. 33–34 in Fig. 5.12). This is the most concise and the most direct presentation of **C5** so far, but it requires familiarity with transitive closure. In order to compute a transitive closure, we need a relation that has the same set as its domain and image. This is intuitively expected: we are supposed to explore the successor relation for states, which defines how to advance from a state to a state; a binary relation on states, a subset of the Cartesian product `State`×`State`. There is no such relation in the model, where connections between states always go via a `transition` object.

In order to derive the successor relation we need to combine two relations together: first choose a transition (`leavingTransitions`), then go to a target state (`target`). Recall that `leavingTransitions` relates states and transitions. It is a subset of `State`×`Transition`. Similarly, `target` is a relation between transitions and states, a subset of `Transition`×`State`. The navigation dot operator in Alloy, as in `leavingTransitions.target`, is implemented as a relational join that "forgets" the internal columns. A standard relational join of these two relations would give a subset of

$$\texttt{State} \times \texttt{Transition} \times \texttt{State} \tag{5.24}$$

In Alloy, the middle column is erased when composing joins, so we obtain a relation which is a subset of the product `State`×`State`. The navigation join gives a relation that is suitable for computing a transitive closure. The intermediate transition objects have disappeared. Let us formalize this. Let the bow tie symbol (⋈) denote a forgetting join:

$$R \bowtie Q \quad \equiv \quad \{(r,q) \mid \text{there exists } p \text{ such that } (r,p) \in R \text{ and } (p,q) \in Q\} \tag{5.25}$$

Then the reflexive transitive closure of `leavingTransitions.target` can be described as:

$$\begin{aligned}&\texttt{*(leavingTransitions.target)} = \\ &\bigcup_{i=0}^{\infty} \underbrace{\texttt{leavingTransitions.target} \bowtie \cdots \bowtie \texttt{leavingTransitions.target}}_{i \text{ times, forget internal columns}}\end{aligned} \tag{5.26}$$

The joint product of zero components above ($i = 0$) is interpreted as the identity relation, the smallest reflexive relation on `State`. Compare the definition in Eq. (5.26) with the definition of transitive closure using predicates in Eq. (5.23), p. 155. The infinity in the definition should not scare you. For finite relations (instances in Alloy are always finite and bounded), the union will stabilize (reach a fixed point) after a finite number of iterations.

As you guessed from the above, in Alloy, `*R` denotes a reflexive transitive closure of relation `R`. We can finally read lines 33–34 in Fig. 5.12. The constraint states that the set of all states is the set of all reachable states, so states that can be reached from `m.initial` by transitive closure of the relation `leavingTransitions.target`.

There is more to Alloy than a brief section can show. We have only used the universal quantifier `all` in our examples. Alloy provides several other quantifiers, including `some` (existential), `one` (exists precisely one), `lone` (at most one), and `none` (exist no). There is also a host of set and relation operations. Constraints can be placed in the context of signatures (like with OCLinEcore), which allows us to eliminate some quantifiers and shorten navigations. All these contribute to extreme brevity of relational constraints. Clearly, Alloy offers the most concise notation of all those discussed above.

> **Exercise 5.14.** To appreciate how context changes formulation, move Constraint **C5** to the context of `FiniteStateMachine` and eliminate the use of the quantifier. To add a constraint to the context of a signature place a Boolean predicate in its own braces directly after closing the signature, without any keyword, as in: `sig ... { attributes here } { constraints here }`.

> **Exercise 5.15.** Implement Constraint **C6** in Alloy. **Hint:** Since all navigations in Alloy compute sets, this can be done by comparing the size of the set of transitions leaving a state with the size of the set of the labels on these transitions.

Alloy tools provide automatic instance generation and visualization. Automatic generation of diverse instances of the model can be used to debug your designs interactively. Jackson [12] recommends generating and reviewing instances for partial designs every time you make changes in a model. Analyzing them often uncovers misconceptions and omissions in the model. For example, the instance in Fig. 5.13 demonstrates that the containment constraints in the first 20 lines of our model are not sufficient. A designer discovering this instance would be compelled to add additional partonomy constraints, as we indeed did in lines 22–26. If no instance can be generated, then this means that our model is *over-constrained*. The constraints are inconsistent with each other and need to be corrected.

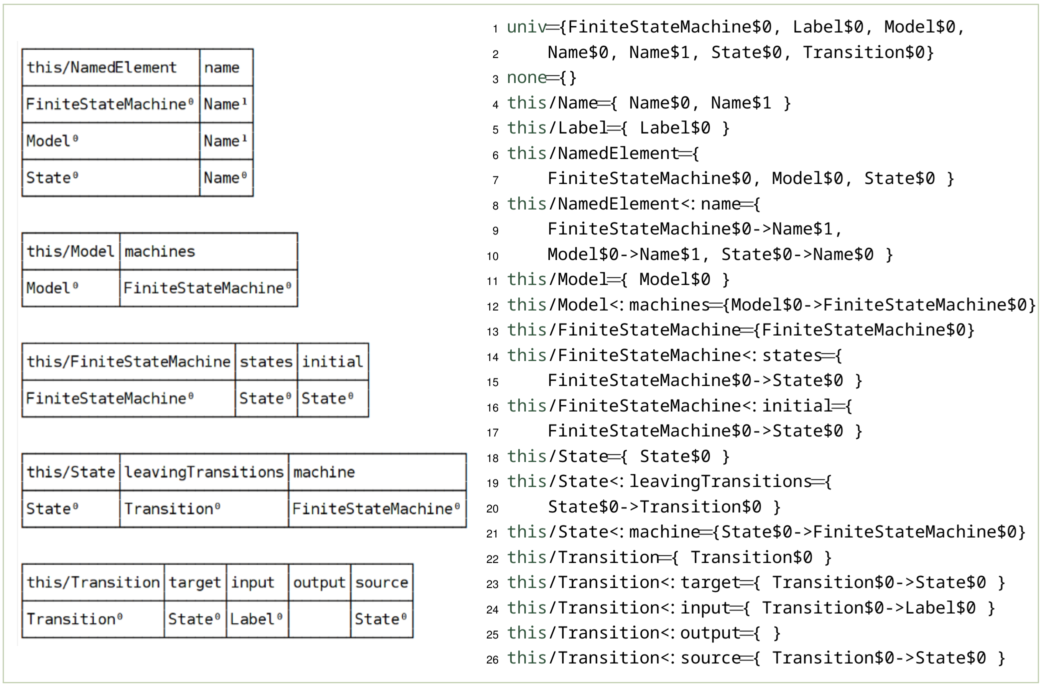
Alloy's analyzer establishes *global consistency* of the model (cf. Def. 3.4). However, we can also use Alloy to establish *element consistency*. In this case, just add an instantiation constraint for the type of the tested element. For example, if we need to generate a model with at least one transition we add the following new constraint to the set of constraints:

```
{ some Transition }
```

Technically, Alloy's tools do not solve the general consistency problem but a *bounded* variant:

Definition 5.27. *A model is* consistent up to a bound *k (k-consistent for short) if and only if there exists a valid instance of size at most k. A model is k-inconsistent if it is not k-consistent.*

There exist *k*-inconsistent models that are consistent in general. Thus Alloy tools can report inconsistency even for valid models. However, Alloy's designers hypothesize that lots of modeling problems can be debugged on very small instances. This is also our experience. In practice, one typically configures Alloy tools with rather small bounds (small *k*) and increases them as needed. This also makes the tools faster. The bound is specified as part of the query to the analyzer.



this/NamedElement	name
FiniteStateMachine^0	Name^1
Model^0	Name^1
State^0	Name^0

this/Model	machines
Model^0	FiniteStateMachine^0

this/FiniteStateMachine	states	initial
FiniteStateMachine^0	State^0	State^0

this/State	leavingTransitions	machine
State^0	Transition^0	FiniteStateMachine^0

this/Transition	target	input	output	source
Transition^0	State^0	Label^0		State^0

```
univ={FiniteStateMachine$0, Label$0, Model$0,
    Name$0, Name$1, State$0, Transition$0}
none={}
this/Name={ Name$0, Name$1 }
this/Label={ Label$0 }
this/NamedElement={
    FiniteStateMachine$0, Model$0, State$0 }
this/NamedElement<: name={
    FiniteStateMachine$0->Name$1,
    Model$0->Name$1, State$0->Name$0 }
this/Model={ Model$0 }
this/Model<: machines={Model$0->FiniteStateMachine$0}
this/FiniteStateMachine={FiniteStateMachine$0}
this/FiniteStateMachine<: states={
    FiniteStateMachine$0->State$0 }
this/FiniteStateMachine<: initial={
    FiniteStateMachine$0->State$0 }
this/State={ State$0 }
this/State<: leavingTransitions={
    State$0->Transition$0 }
this/State<: machine={State$0->FiniteStateMachine$0}
this/Transition={ Transition$0 }
this/Transition<: target={ Transition$0->State$0 }
this/Transition<: input={ Transition$0->Label$0 }
this/Transition<: output={ }
this/Transition<: source={ Transition$0->State$0 }
```

Figure 5.14: An example instance generated by Alloy tools for the model in Fig. 5.12. Presented as relations in the left column, and using Alloy's textual instance syntax in the right column

The limitation of Alloy to bounded problems follows from the underlying reasoning technology: predominantly SAT-solving. Variants of Alloy and similar languages based on Constraint Programming (CP) and Satisfiability Modulo Theory (SMT) solvers also exist. However, all these reasoners, even though very fast, can only represent fixed-size problems.

Alloy tools present instances as graphs (Fig. 5.13) or tables. The table presentation may clarify the relational nature of the language. The left column in Fig. 5.14 shows the tables for a small instance containing one model object, with one state machine that contains a single state with a single loop transition. The example instantiates each class of the partonomy in Fig. 3.4 once. The format uses one table per signature. The first column in each table is the primary key. The remaining columns are foreign keys referring to other tables. We encourage the reader to reconstruct the structure of the instance on paper, from this representation. Tables for names and labels (both single column) are omitted for brevity.

Clearly, Alloy is not a static-semantics definition language in the same way as OCL and the GPLs used earlier in this chapter. There is no way to attach its constraints to syntax trees in Ecore or to types in programming languages. Only meta-models written in Alloy can be constrained, and there is no easy way to develop languages on top of these meta-models. However, one can translate models from other languages into Alloy syntax, and use Alloy tools to evaluate constraints on them. Once instances are found, the

Alloy output can be parsed and translated into whatever technical space you need. Since Alloy is a solver, not an evaluator, a whole range of more powerful checks can be implemented than when using other languages.

The right panel in Fig. 5.14 shows the same example as in the left panel, but in Alloy's textual syntax for instances. This format is rather easy to parse. If we translated it to XMI, Scala, or our concrete syntax for state machines, we could use the automatically generated instances to test the tool chain of our language, including generators and interpreters. If we had built a different Alloy model capturing the execution traces of a state machine, then the instances could represent sequences of inputs to the machine. After loading these sequences in a Java program, we could use them to drive automatic testing tools for an `fsm` interpreter.

Finally, we can also use Alloy for simple program or model synthesis. One needs to build a model that describes programs of interest with constraints. For our example, if we add the following constraint in Alloy, the tools will synthesize a simple state machine of the shape resembling a coffee machine, entirely automatically.

> A model with a single state machine that owns two states connected with exactly two transitions in a cycle labeled by `'coin'` and `'coffee'`.

Exercises 3.34 (p. 84) and 4.57 (p. 141) explore parsing of Alloy outputs.

This method has been used in research for many DSLs. Over the years, researchers have exploited the expressive and clean Alloy syntax to built tools supporting specialized constraint languages beyond instance finding and consistency checking. The applications include testing, synthesis, diagnostics and repair, simplification, model merging, etc. Targeting Alloy as a solver (instead of the more basic formats of SAT, SMT, and CP solvers) speeds up tool development considerably. If you have a constraint-modeling and solving mindset, and know Alloy, you will easily find tasks around your language that can be automated.

5.4 Guidelines for Writing Constraints

Constraints can be tricky to write. Let us discuss good practices in constraining syntax.

General Hints for Writing Static Semantics Constraints

Guideline 5.1 *Consider not defining the static semantics at all!* This may sound crazy after reading three sections arguing exactly the opposite! However, side-stepping constraint and type system definition (Chapter 6) is often a natural choice. For minimalistic languages, developed on a tight budget, it may make sense to never develop a static semantics validator. If your language is based on code generation you may be able to *piggy-back on your target language* [15]. Imagine, for instance, that we generate C code from finite-state-machine

Constraints as a Modeling Paradigm

Constraints are the basis of a very useful paradigm in modeling, exercised at its full in tools from the *Constraint Programming* community [6, 19]. Many constraint languages derive from first-order logic and relational algebras, exploiting the result of Edgar Codd (1923-2003) that first-order logic adequately captures relational (so structural) modeling. The modeling mindset is to restrict the infinite set of graphs to only those of interest. Alternatively, we think in terms of graphs that should be generable by an instance generator. Constraint modeling is a strong form of declarative programming: you state requirements (the "what") and delegate finding solutions, proving consistency, or verifying facts that hold in a model to a solver (the "how" and "why").

In this chapter, you have seen constraints written in English, in first-order logic, in a range of programming languages, in OCL, and in Alloy. We hope that this exposed you to the constraint-modeling paradigm sufficiently to recognize constraint writing as a specialized but useful modeling skill that can help to solve a range of problems, even outside static semantics. A skill, a mental model even, that carries beyond the concrete languages used as examples here.

```
switch (input) {
  case COFFEE:
    next_state = BREWING;
    break;

  case COFFEE:
    next_state = PAYMENT;
    break;
```

```
double-case.c: In function ‘main’:
double-case.c:14:5: error: duplicate case value
     case COFFEE:
     ^~~~
double-case.c:10:5: note: previously used here
     case COFFEE:
     ^~~~
```

Figure 5.15: ***Left:*** *A hypothetical code generated from a non-deterministic state machine violating Constraint* ***C6***. ***Right:*** *A compiler error from GCC that can serve as "piggy-backing." An actual generator for* `fsm` *is shown in Fig. 9.9 on p. 333*

models (we will indeed generate such code in later chapters). If our machine has non-deterministic transitions, violating Constraint **C6**, we might produce code like that presented in the left part of Fig. 5.15: a switch statement with a duplicate entry. This in turn can cause the C compiler to complain, as shown in the right part of the figure. If this kind of error is acceptable to your users, you may choose not to implement the constraint validation at all!

Define static semantics later rather than earlier. Even if you decide that a static checker is needed for your DSL, there are several advantages to delaying its design. In agile development, it is important to scaffold a working tool chain as early as possible, in very few sprints. This way your users can start to experiment with it. You can receive early feedback. They can start advancing their projects. Early on, it is less important how the tools will behave on ill-defined inputs. Early iterations of language tools do not need to be tested on invalid models. Good diagnostics, error detection, and reporting can often be designed in later iterations, when the scope and design of the language are stabilizing, and the rest of the tool chain is maturing. ***Guideline 5.2***

We recommend to design an interpreter (Chapter 8) or a code generator (Chapter 9) first, achieving a minimal viable prototype of the language as early as possible. A set of constraints is primarily defined to capture the assumptions of the dynamic semantics. These assumptions will be much better understood when you already have dynamic semantics! This is even more important for type checkers (Chapter 6): a type checker and the associated type system are really an approximation of dynamic execution. It is essentially impossible to come up with a good and useful approximation without a good understanding of the dynamic semantics itself. Thus, we recommend implementing a type checker after completing at least the first iteration of an interpreter or a code generator.

Guideline 5.3 *A general-purpose or a specialized constraint language?* Should I follow the advice of Sect. 5.2 or of Sect. 5.3? Should I use an external tool like Alloy? Shall I design all constraints in basic mathematics first, like in Sect. 5.1, and only then rewrite to a programming language?

The decision between specialized and general-purpose languages hinges on the trade-off between easy access to programming experts and the need to use specialized reasoning tools. If the goal is to validate inputs, and there is no need for instance generation, choose the language which will be easy to work in for you and for others around you. This way the constraints can easily change ownership to new programmers and the project can live longer. This will typically be the same language in which you implement the rest of your DSL tools. Do not work on paper, or not for too long. Implement constraints in a programming language and start running and testing them iteratively as soon as you can, even before a constraint formulation is finalized. (We did not endorse mathematical logic as a software development mechanism, but as a way to introduce you to thinking in terms of constraints.)

Specialized constraint languages are useful in two scenarios: if reasoners are needed for specialized applications, and if you are planning to develop many languages. In the former case, you have little alternative. In the latter, for example if you work for a language consultancy, the increased conciseness and readability of constraints will pay over time for the steeper learning curve and more cranky tools.

Guideline 5.4 *Maximize the diagram, minimize the constraints.* A class diagram is a constraint system itself, just of limited expressiveness. As already discussed in Sect. 5.1, many of the same constraints can be stated both in the constraint language and in the diagram language. When the partonomy constraints (black diamond) turned out not to be supported in Alloy, we expressed the same semantics using relational constraints (lines 22–26 in Fig. 5.12). Conversely, instead of using diagram annotations in Ecore, we could have written textual constraints to limit cardinalities of associations, or to bind two unidirectional references into a bidirectional one.

Exercise 5.16. For the following constraints discuss how they could be enforced using an appropriate construct in an Ecore meta-model instead:

a) A constraint written in Scala for the meta-model in Fig. 5.1:

```
inv[Person] { _.getChild.size <=2 }
```

b) A constraint written in OCL for the finite-state-machine meta-model:

```
    invariant: self.name.notEmpty ()
```

Whenever there is a choice, it is advisable to express syntactic restrictions in the meta-model, leaving only the impossible to the textual constraints. Diagrammatic idioms are easier to recognize and to explain to other developers. Recognizable diagrammatic patterns may appear very cryptic in a textual language (compare the black diamonds in Fig. 3.1 with the corresponding five constraints in Alloy in Fig. 5.12). If you incorporate diagrammatic idioms into a meta-model, you can expect better diagnostics of instances from the modeling framework, and better type checking and runtime checking of your language implementation code. This is because the generated meta-model code that you are programming against will be aware of these constraints, and reflect them in types and runtime checks as appropriate.

This guideline should be applied pragmatically. Sometimes, maximizing a diagram is not the best strategy. If expressing a constraint in the diagram requires significant refactoring, introducing auxiliary classes, or splitting classes into sub-concepts that do not reflect the concepts in the domain, then declaring constraints outside the diagram is preferable. The bottom line is that you should try to use structural notations, such as class diagrams or ADTs, primarily for representing fundamental intuitive structural constraints. Non-standard intricate restrictions should be specified separately in a logic-based formalism.

Guideline 5.5 *Static semantics depends on abstract syntax alone. Refer only to the model properties in a constraint.* Even if written in a GPL, constraints are conceptually a part of the meta-model. The only data that you can refer to from constraints are model properties. Nothing else. Constraints are used to enforce integrity of the model itself. It is a common encapsulation failure to bring other concerns to constraints. This makes them difficult to test in isolation from the rest of the system, and hard to reuse in new tools for the DSL. None of our examples, even in Scala or C#, has violated this assumption (cf. Tables 5.2 and 5.3, and Figures 5.10 and 5.12).

A common violation of this guideline is an introduction of dependencies between the Eclipse IDE platform (or another editor) and the validity checker, for instance when producing error messages. A static checker that depends on an IDE platform is extremely difficult to use in a standalone command-line or server-side tool, which you will presumably need sooner or later. Even if you succeed in embedding the IDE with the static checker, the executable will be far from lean and nimble. The large dependency will make it brittle, susceptible to co-evolution problems with the related big piece of software. A careful reader will notice that the example code for this book has been carefully designed to avoid such dependencies, even in the parts where we use the Eclipse Modeling Framework and Xtext.

If you absolutely need to refer to other parts of the system, if the model needs to be validated for consistency with other files, for instance database entries or configuration files, do this validation in another pass. Encapsulate

```
 1 val C2_GOOD = inv[FiniteStateMachine] { self =>
 2   self.getStates.forAllDifferent { (s1,s2) => s1.getName != s2.getName }}

 4 def C2_BAD (m: FiniteStateMachine): Boolean =
 5   var it1 = m.getStates.iterator
 6   while it1.hasNext do
 7     var s1: State = it1.next
 8     var it2 = m.getStates.iterator
 9     while it2.hasNext do
10       var s2: State = it2.next
11       if s1 != s2 && s1.getName == s2.getName then
12           return false
13   return true
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/constraints.scala

Figure 5.16: ***Top:*** *(l. 1–2) Our favorite declarative formulation of* ***C2*** *in Scala. See Exercise 5.22 about* `forAllDifferent`. ***Bottom:*** *(l. 4–13) A verbose and lengthy imperative formulation of* ***C2****, typical of inexperienced constraint writers. Avoid this! Both examples are written in Scala, but the problem is seen in most languages*

the meta-model constraints in a self-contained module, only dependent on the abstract syntax, and interface to that module from a bigger checker.

Programming Constraints and Avoiding Bad Smells

Guideline 5.6 *Keep constraints declarative, as close to natural language as possible.* There is clearly a correspondence between requirements written in English and in formal logic and programming languages. To avoid obfuscating this correspondence, keep your constraints as declarative as possible, as close to natural language as possible, devoid of low-level computational primitives such as variable assignments, loops, or return statements (Fig. 5.16). Implement iteration with higher-order functions instead. Extend your constraint language with needed operators and iterators (implication, *n*-ary quantifiers, all-different, etc.). You will quickly accumulate a useful vocabulary of primitives. Finally, if you really need recursion or loops (for instance to implement transitive closure), encapsulate the necessary computations in Boolean predicates, to be able to use them in other declarative constraints.

Guideline 5.7 *Do not modify the model from the constraint code.* We argued already against side effects in constraints. Constraints can be checked multiple times from within the modeling environment, and in other tools. They can be constantly evaluated by the modeling editor to provide live feedback. The user may not control these checks explicitly. So these executions should not have any visible side effects on the model. Specialized constraint languages are designed to avoid such mistakes. Exercise extra care when using a GPL.

The only exception to this guideline is when a static-semantics device (most typically, but not only, a type system) infers new information about the model. It might be practical to augment (decorate) the model with new values, but never change the existing ones. For instance, we may want to annotate sub-expressions with inferred types. In any case, this should be done in such a way that the checkers can be run multiple times without harm.

Guideline 5.8 *Avoid top-level conjunction.* It is tempting to combine several different aspects in a single constraint using logical conjunction. Avoid this though. One constraint should capture one English sentence, as simply as possible. There is rarely advantage to creating compound constraints. Small atomic

```
inv[Student] { self =>
    !self.getName.isEmpty && self.getAge >= 18 }
```
Error: An empty name or the age is below 18!

```
inv[Student] { !_.getName.isEmpty }
```
Error: An empty student name.
```
inv[Student] { _.getAge >=18 }
```
Error: The age is below 18.

Figure 5.17: ***Top:*** *a compound constraint that mixes age and name aspects that are otherwise unrelated.* ***Bottom:*** *The same constraint split into two separate ones. Observe how the error message for the user becomes more precise with the split*

```
context Model
invariant: Transition.allInstances->forAll (t|t.source<>t.target)

context Transition
invariant: source <> target

context FiniteStateMachine
invariant C3: State.allInstances ()->exists (s |
                  self.states->includes (s) and self.initial=s)
```

Figure 5.18: ***Top:*** *Using OCL's* `allInstances` *to forbid self-loop transitions (*`Transition`*).* ***Center:*** *Replace* `allInstances` *with a better context.* ***Bottom:*** *A needlessly convoluted formulation of Constraint* ***C3*** *using* `allInstances`*. Contrast with Fig. 5.10*

independent constraints map to precise and informative error messages for the users. If a compound constraint fails, it is difficult to see which part of the conjunction is invalid, leading to vague error messages (Fig. 5.17).

Guideline 5.9

Avoid quantifications over all instances of a type. In OCL, `T.allInstances` evaluates to all objects of type `T`. GPL libraries for computing over abstract syntax also provide a similar reflection capability. Figure 5.18 shows an example of a constraint on finite-state machines forbidding self-loop transitions (so transitions with source and target in the same state). The first variant uses iteration over all instances of a type. The second variant eliminates the use of this very general construct by placing the transition in the context of a `Transition` object. The bottom part of the example shows Constraint **C3** written in the context of class `FiniteStateMachine` (like our original example), but using iteration over all instances of type `State`. This formulation is much more complex than our original proposal in Fig. 5.10.

The use of `allInstances` should be avoided in favor of a good context class for the constraint. Iteration over all objects of a type tends to be computationally expensive and lowers understandability of the constraints—it escapes from the instances to the meta-level. The method `allInstances` is static, so it belongs to the meta-class, not to the instances. If you need to get all instances of more than one class, you will be able to do it by navigating from the document root (or any closer class), as discussed below.

Guideline 5.10

Refactor constraints to the optimal context. Constraints placed in a wrong context require long navigations and many quantifiers. In Fig. 5.19, the class `Model` is a worse choice for the context of **C3** than the class `FiniteStateMachine`, which we chose originally. Shifting the context a level upwards along the partonomy costs an additional universal quantifier.

Figure 5.19: ***Top:*** *Constraint* ***C3*** *in the optimal context of a machine.* ***Bottom:*** ***C3*** *in the context of a* `Model`. *See also another example in Eq. (5.20) on p. 153*

```
inv[FiniteStateMachine] { self =>
  self.getStates.contains (self.getInitial) }

inv[Model] { self =>
  self.getMachines.forall { fsm =>
    fsm.getStates.contains (fsm.getInitial) } }
```

Table 5.5: *Several examples of eyebrow-raising expressions that should be simplified in static semantics constraints (and pretty much in any other kinds of computer programming). Examples are presented in Scala syntax, but most apply to other languages*

Do not write	**Write instead**
`if e then true else false`	`e`
`if e then false else true`	`!e`
`if e1 then e2 else false`	`e1 && e2`
`if e1 then true else e2`	`e1 || e2`
`if e1 then e2 else true`	`e1 implies e2`
`if e1 then true else e2`	`!e1 implies e2`
`e && true`	`e`
`false || e`	`e`
`!e1 || e2`	`e1 implies e2`
`!(!e)`	`e`
`return e;`	`e`

Exercise 5.17. Besides long navigations and excess quantifiers, inability to navigate to the set of instances needed in a constraint also indicates a possibility of the wrong context type. If you need to resort to `allInstances` (in OCL) to work around such problems, the context is likely incorrect, too. To appreciate this issue re-write **C3** in the context of `NamedElement` of the meta-model in Fig. 3.1.

It takes experience to select the right context. In sequences of universal quantifiers, check to see whether any prefix of them can be avoided by shifting the context to the type one of them ranges over. If long navigations descend down the partonomy, especially starting with the same prefix, you are likely using a context too high in the partonomy hierarchy. Check if any objects down in the navigation prefix would not be a better candidate. Dually, if long navigations go upwards in the partonomy, it might be that the context needs to be moved higher. Beware though that these rules are not strict. You will meet long navigations in constraints that are *inherently non-local*. There is not much you can do about this, except perhaps to revisit the meta-model design and add explicit or derived associations. A constraint relating very remote classes may be an indicator of a missing association in the model.

Guideline 5.11 *Avoid verbose Boolean constructs, especially using true and false.* Programmers inexperienced in functional style, which is so strongly exercised in constraints, tend to write expressions that are suboptimal in subtle but irritating ways. Table 5.5 lists some common examples. Chiefly, an appearance of a constant `true` or `false` in a Boolean expression is a bad-smell. Most often the constant can be eliminated in favor of applying a simpler operator that more directly states the intention. This way the constraint's intention is more easily available to the prospective reader of the code.

5.5 Quality Assurance and Testing for Static Constraints

Front-end failures are visible to the language users, so the corresponding bugs tend to be detected early and fixed quickly. Nevertheless, experienced language engineers admit that this does not prevent many errors from hiding. Ratiu, Völter, and Pavletic [18] describe an extensive case study in which they found many bugs in a collection of DSLs, despite high confidence in the implementations a priori. They used automated random testing to test murky cases. Indeed, some aspects of requirements-based white-box scenario tests for constraints can be automated using randomization.

Scenario testing. We begin with an exploratory question.

> **Exercise 5.18.** Constraint **C3** (p. 146) requires that *the initial state of each state machine is one of its own states.* We formalized this in Tbl. 5.2, p. 156. Create an instance of the `fsm` meta-model (Fig. 3.1, p. 53) which violates this constraint.

A buggy constraints checker not only admits semantically meaningless models, but it may also prevent users from creating legitimate ones. Scenario-based testing of constraints aims to establish whether the right instances are admitted and rejected by our implementation. Test at least one positive and at least one negative scenario for each constraint. At least one of your tests should be passing when the constraint holds and at least one when the constraint is *not* satisfied. Do not ignore the negative tests: they ensure that a constraint catches input errors effectively, so they test its main purpose!

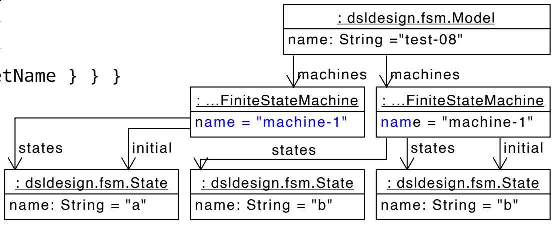
We illustrate this with an example, creating tests for Constraint **C2** from p. 146: *all states within the same machine must have distinct names.* Figure 5.20 recalls the constraint in the top left corner. The same constraint in other programming languages has been shown in Tbl. 5.3 on p. 160. The main part of Fig. 5.20 shows five example test cases, each a single tree in the instance specification notation of UML. We number these test cases for reference; find the numbers in the `name` attribute of each of the five root `Model` objects. The figure shows four positive and one negative test cases:

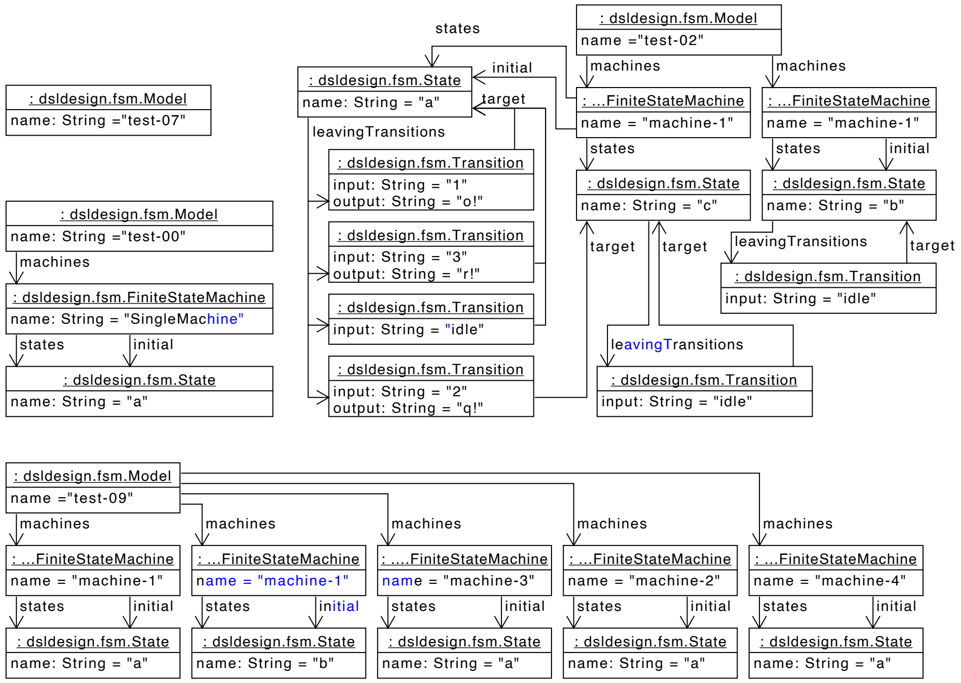
`(+) test-00:` a single machine with a single state
`(+) test-02:` two machines with a name clash, each with a single distinct state
`(+) test-07:` an empty model, no machines, no states
`(+) test-09:` five machines with some name clashes, each has a state named `"a"`
`(-) test-08:` two machines, one of them has two states named `"b"`

Convince yourself that **C2** indeed passes (respectively fails) on these inputs. Simultaneously, observe that these test cases are selected specifically for **C2**, which allows for good fault localization and exploring various aspects of the constraint. The positive tests do not necessarily validate with other constraints. For instance, `test-02` violates Constraint **C1** that *all machines must have distinct names*, yet we use it as a positive scenario for **C2**.

Test cases are best stored in abstract syntax. Since this example is developed with Ecore, we used XMI files—the UML notation was only useful for creating a figure. The bottom part of Fig. 5.20 shows how files

```
val C2 = inv[FiniteStateMachine] { m =>
  m.getStates.asScala.forall { s1 =>
  m.getStates.asScala.forall { s2 =>
    s1!=s2 implies s1.getName!=s2.getName } } }
```

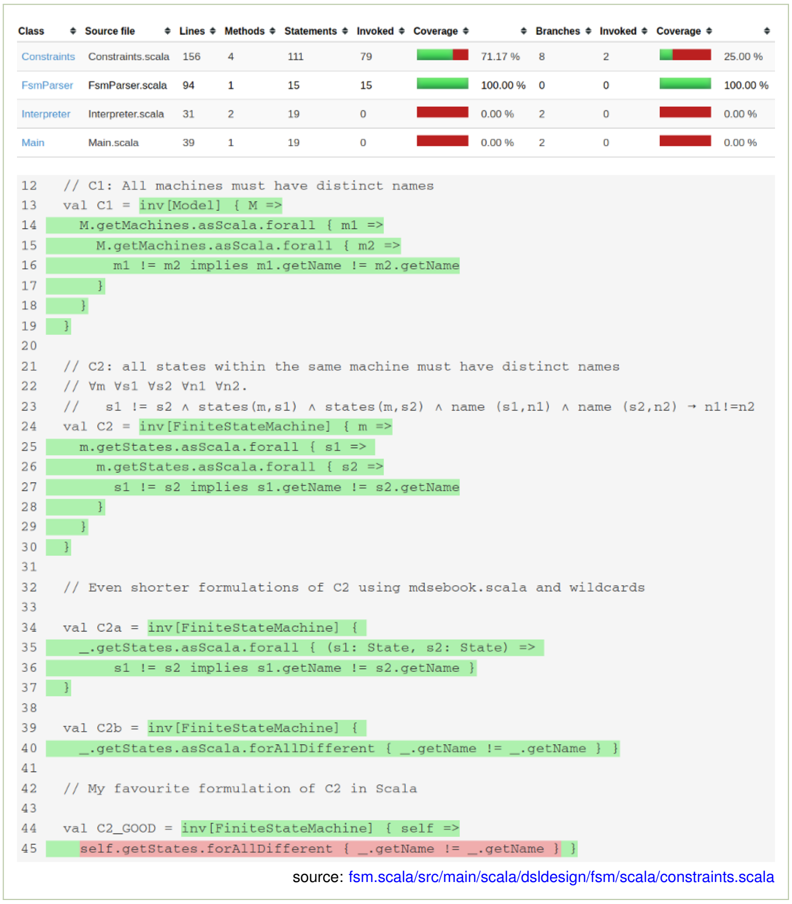




```
def load (name: String) =
  EMFScala.loadFromXMI[dsldesign.fsm.Model] (s"../dsldesign.fsm/test-files/${name}.xmi")

"positive tests" - {
  "single machine (test-00)"      in { C2.checkAll (load ("test-00")) shouldBe true }
  "two larger machines (test-02)" in { C2.checkAll (load ("test-02")) shouldBe true }
  "empty model (test-07)"         in { C2.checkAll (load ("test-07")) shouldBe true }
  "names across scopes (test-09)" in { C2.checkAll (load ("test-09")) shouldBe true }
}

"negative tests" - {
  "two machines (test-08)" in { C2.checkAll (load ("test-08")) shouldBe false }
}
```

source: fsm.scala/src/test/scala/dsldesign/fsm/scala/constraints/ConstraintsSpec.scala

Figure 5.20: ***Top:*** *Test cases for Constraint* ***C2****: four positive (nos. 0, 2, 7, and 9) and one negative (no. 8). Find the numbers of the test cases in the name attribute of the* `Model` *objects. The XMI files can be found in the code repository at fsm/test-files/.* ***Bottom:*** *The Scala test code for these examples*



Class	Source file	Lines	Methods	Statements	Invoked	Coverage	Branches	Invoked	Coverage
Constraints	Constraints.scala	156	4	111	79	🟩🟥 71.17 %	8	2	🟩🟥 25.00 %
FsmParser	FsmParser.scala	94	1	15	15	🟩 100.00 %	0	0	🟩 100.00 %
Interpreter	Interpreter.scala	31	2	19	0	🟥 0.00 %	2	0	🟥 0.00 %
Main	Main.scala	39	1	19	0	🟥 0.00 %	2	0	🟥 0.00 %

```
  // C1: All machines must have distinct names
  val C1 = inv[Model] { M =>
    M.getMachines.asScala.forall { m1 =>
      M.getMachines.asScala.forall { m2 =>
        m1 != m2 implies m1.getName != m2.getName
      }
    }
  }

  // C2: all states within the same machine must have distinct names
  // ∀m ∀s1 ∀s2 ∀n1 ∀n2.
  //   s1 != s2 ∧ states(m,s1) ∧ states(m,s2) ∧ name (s1,n1) ∧ name (s2,n2) → n1!=n2
  val C2 = inv[FiniteStateMachine] { m =>
    m.getStates.asScala.forall { s1 =>
      m.getStates.asScala.forall { s2 =>
        s1 != s2 implies s1.getName != s2.getName
      }
    }
  }

  // Even shorter formulations of C2 using mdsebook.scala and wildcards

  val C2a = inv[FiniteStateMachine] {
    _.getStates.asScala.forall { (s1: State, s2: State) =>
        s1 != s2 implies s1.getName != s2.getName }
  }

  val C2b = inv[FiniteStateMachine] {
    _.getStates.asScala.forAllDifferent { _.getName != _.getName } }

  // My favourite formulation of C2 in Scala

  val C2_GOOD = inv[FiniteStateMachine] { self =>
    self.getStates.forAllDifferent { _.getName != _.getName } }
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/constraints.scala

Figure 5.21: A fragment of a test-coverage report for tests of constraints for the `fsm` example; ***Top:*** a summary of the statistics, ***Bottom:*** detailed coloring of covered and not covered code

are loaded and included in Scalatest's test. Had we not used Ecore, the test cases in abstract syntax could have been stored as plain values in a GPL, like we did in Fig. 3.7, or in another standard format for structured data, for instance JSON or YAML. By using abstract syntax to represent test cases we decouple the implementation of static checking from concrete syntax. You can test the constraints before the parser for the language is ready, which is a common practice in many language projects.

If a static checker is decomposed into many simple independent constraints, it may suffice to consider just two test cases for each. Complex constraints, however, may require many test cases to activate all subexpressions. To independently verify that we have tested all constraints, we can use a test-coverage tool. The top part of Fig. 5.21 shows a report for our constraints from `scoverage`,[10] covering 71.17% of statements. We do not reach 100%, as we only implemented tests for Constraints **C1**–**C6**, while

[10] https://github.com/scoverage/ is a tool for Scala (2022/09). Similar tools exist for other programming languages—another good reason to implement constraint checking in a GPL

```
 1 def Model (name: String, machines: Seq[FiniteStateMachine]=Nil): Model =
 2   val model = factory.createModel
 3   model.setName (name)
 4   model.getMachines.addAll (machines.asJava)
 5   model

 7 def FiniteStateMachine (name: String, states: Seq[State], ini: State) ...

 9 val genName: Gen[String] = for
10   L <- Gen.choose (1, 30)
11   first <- Gen.alphaChar
12   chars <- Gen.listOfN (L, Gen.alphaNumChar)
13 yield (first ::chars).mkString

15 val genModel: Gen[Model] = for
16   name <- genName
17   M <- Gen.choose (0, 5)
18   machines <- Gen.listOfN (M, genMachine)
19 yield Model (name, machines)

21 val genMachine: Gen[FiniteStateMachine] = for
22   name <- genName
23   S <- Gen.choose (1, 10)
24   states <- Gen.listOfN (S, genState)
25   initial <- Gen.oneOf (states)
26   T <- Gen.choose (0, 20)
27   transitions <- Gen.listOfN  (T, genTrans (states, genName))
28 yield FiniteStateMachine (name, states, initial)

30 val genState: Gen[State] = for name <- genName yield State (name)

32 def genTrans (states: Seq[State],gen: Gen[String]): Gen[Transition] = for
33   input <- gen
34   output <- gen
35   source <- Gen.oneOf (states)
36   target <- Gen.oneOf (states)
37 yield Transition (input, output, source, target)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/generators.scala

Figure 5.22: *Generators of random correct state machines that should validate positively. Lines 1–7 show helper factory functions, and lines 9–37 show the actual generators using Scalacheck's* `Gen` *API*

the test project contains other examples and code that have not been tested at the time of the report generation; for instance alternative implementations of **C2** used in this chapter. Unfortunately, branch coverage is very unreliable for constraints (25%). Coverage tools typically do not detect branches at the expression level. Branches tend to be defined by conditional statements and loops, which does not agree well with the functional style of meta-model constraints. The `scoverage` tool only finds eight branches in the entire module (!), most in the code we have not tested. It is useful to inspect the detailed coverage report. A fragment is shown in the bottom of Fig. 5.21. There we can see that **C1** and **C2** have been executed; uncovered code is highlighted in red.

Automating random testing for constraints. In order to automate test selection, we generate model instances randomly. This can be conveniently done using the generator support from property-based testing libraries such as Quickcheck for Haskell [3], JUnit-Quickcheck for Java, Hypothesis for Python, and FsCheck for F#. We demonstrate the idea using Scalacheck. Scalacheck uses a generic type to represent generators. A value of type `Gen[A]` is a generator of values of type `A`. Scalacheck provides ready-made generators for standard types and an API for constructing complex generators from simple ones. For instance, `choose (0, 5)` is a `Gen[Int]` which produces integer numbers from the interval $[0;5)$, while `listOfN[A](42,g)` is a `Gen[List[A]]` producing lists of up to 42 elements populated by generator `g`, if the type of `g` is `Gen[A]`.[11]

Generating random abstract-syntax trees structurally resembles parsing, as it is, in a way, a reverse operation. We instantiate syntax types using random values from `Gen` instead of using the structure of an input string. An example is found in Fig. 5.22. Lines 1–7 define helper factory functions, used below to build syntax trees. The first creates a `Model` node with the given name and a set of machine objects. The second creates a finite-state machine with a name, a set of states, and an initial state. We create such convenience factories for all concrete classes of Fig. 3.1 (not shown). Had we used abstract syntax defined as an algebraic data type (Fig. 3.5), we would just invoke the type constructors directly. We use the wrappers here to hide the chatty style of Ecore from the functional API of generators.

The main part of the example is found in lines 9–37. The generator `genName` first decides the length of a name (l. 10), then picks a random letter to be the first character (l. 11) and a list of alphanumeric characters (l. 12), and creates a string out of this suffix and the previously selected prefix (l. 13). The generator `genModel` (l. 15–19) builds a model object by creating a random name (`genName`) and adding up to five finite-state-machine objects (l. 17–19). Each state machine (l. 21–28) is populated with up to ten states, one of them designated as initial (a mandatory property in the meta-model). Finally, we generate up to twenty transition objects (l. 26–27) connecting the states of the same machine to create “well-behaved” models. For this we pass a set of states to `genTrans`.

Generation of well-behaved models exploits the partonomy of the meta-model. The more tree-like the abstract syntax, the easier it is to structure the generators. In Fig. 5.22, we descend from models, to machines: states and transitions are generated for each machine separately and connected to the states of only this machine. For meta-models with complex structures and dependencies, generating type-correct instances is difficult. In such a case it might be better to use a solver or a tool like Alloy Analyzer.

Once we have the generators, we can use them in tests. Figure 5.23 shows how to build a simple fuzzer—a tool that feeds random data into a static checker and tests stability, so whether crashes appear. Here we test the EMF validator, so the meta-model constraints; later we will show

[11] Find more generators in the implementation of `Gen` at https://github.com/typelevel/scalacheck/blob/1.14.x/src/main/scala/org/scalacheck/Gen.scala, retrieved 2022/09

Figure 5.23: *A simplistic fuzzer for meta-model constraints. Among other things it tests whether our generator indeed constructs valid instances in the eyes of EMF/Ecore*

```
1 "Instances created by genModel validate" in check {
2   Prop.forAll (genModel) { m: Model =>
3     validate (m)
4       .isEmpty
5       .before { EcoreUtil.delete (m) }
6 } }
```

```
val genFreeName: Gen[String] = for
  predefined <- Gen.oneOf ("Name1" , "Name2", "Name3", "Name4")
  fresh <- Gen.asciiPrintableStr suchThat { !_.isEmpty }
  name <- Gen.frequency (2 -> predefined, 1 -> fresh)
yield name.substring (0, name.size min 60)

val genFreeModel: Gen[Model] = for
  name        <- genFreeName
  numMachines <- Gen.choose  (0, 30)
  machines    <- Gen.listOfN (numMachines, genFreeMachine)
  numStates   <- Gen.choose  (numMachines, 2*numMachines)
  states      <- Gen.listOfN (numStates, genFreeState (machines))
  numTrans    <- Gen.choose  (0, 10*numMachines)
  // Values below ignored as side effects connect elements to the model
  _           <- Gen.listOfN (numTrans,genTrans (states,genFreeName))
  _           <- Gen.sequence { for ma <- machines
                     yield Gen.oneOf (states).map (ma.setInitial) }
yield Model (name, machines)

def genFreeState (machines: List[FiniteStateMachine]): Gen[State] = for
  name  <- genFreeName
  owner <- machines.find { _.getStates.isEmpty } match
    // prioritize machines with no states as owners
    case Some (m) => Gen.const (m)
    case None     => Gen.oneOf (machines)
yield State (name, owner)

val genFreeMachine: Gen[FiniteStateMachine] =
  for name <- genFreeName yield FiniteStateMachine (name, Nil, null)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/generators.scala

Figure 5.24: *Generators of "free" models that are likely to fail validation with Constraints **C1**–**C6**. We use the same* `genTransition` *function as in Fig. 5.22*

how to test our own constraints. The `validate` call (l. 3) returns an option object, containing diagnostic information. If the option is empty (l. 4), the test passes, as no diagnostics means no failure. Just before finishing the test, we deallocate the model, so as not to pollute the EMF heap (l. 5). The framework tests 100 random inputs by default.

The models generated so far strive to be correct: they should validate successfully with our constraints. Random statically valid models are extremely useful for testing of the later phases of the tool chain, including interpreters and code generators. However, to test constraints we also need negative examples. Figure 5.24 shows another set of generators that are less conservative and create possibly broken "free" machines, which may violate Constraints **C1**–**C6** but which observe the meta-model constraints. Testing constraints requires inputs that validate with EMF or the host language (if using ADTs), as otherwise the test fails even before we try to run a constraint.

```
1  def consistent (c: Constraint): Prop =
2    Prop.exists (genModel) { m: Model =>
3      c.checkAll (m) before { EcoreUtil.delete (m) } }

5  "C1 is consistent"  in check { consistent (C1) }
6  ...
7  "C6 is consistent"  in check { consistent (C6) }

9  def falsfiable (c: Constraint): Prop =
10   Prop.exists (genFreeModel) { m: Model =>
11     (!c.checkAll (m)) before { EcoreUtil.delete (m) } }

13 "C1 can be falsifiable"  in check { falsifable (C1) }
14 ...
```

source: fsm.scala/src/test/scala/dsldesign/fsm/scala/constraints/ConstraintsFuzzSpec.scala

Figure 5.25: *Testing consistency and falsifiability of constraints using random generation of models from Figures 5.22 and 5.24*

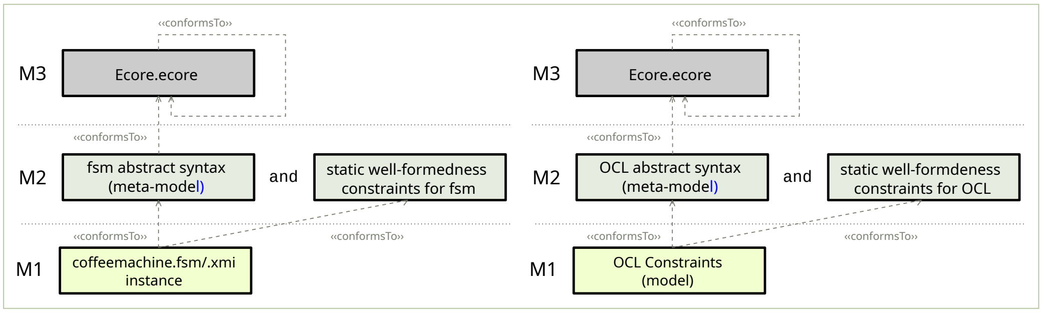
We use the same helper factory functions as in Fig. 5.22. A new generator for names (l. 1–5) ensures that names will contain a wider set of characters (l. 3) while also allowing for some name clashes (l. 2). To allow for violations, we abandon the top-down design and create sets of states and transitions independently of state machines (l. 12 and 15). We then arbitrarily assign them to individual machines (l. 25–26, and 15). This uses the generator of transitions shown in Fig. 5.22 but with a different set of states, now coming from multiple machines. The initial states are selected randomly, disregarding their ownership (l. 16–17). Many of these models should now fail our constraints, so they can serve as both positive and negative test cases.

We will use positive and negative test cases to establish consistency and falsifiability of individual constraints. These concepts are very similar to the concept of consistency of a meta-model (Def. 3.4, p. 69).

Definition 5.28 (Local consistency). *A constraint is* consistent *if there exists a model for which it holds. A constraint is* falsifiable *if there exists a model for which it does not hold.*

Consistency and falsifiability are key internal quality criteria for constraints, independent of the user requirements for the language. An inconsistent constraint in a static checker prevents the language tool from processing *any* models. If a constraint is not falsifiable then it is useless—it does not contribute to the discriminating power of the checker. It cannot separate the good and bad inputs. Whenever you create a positive test case (that passes) you prove that a constraint is satisfiable. Whenever you create a negative test case (that passes) you prove that a constraint is falsifiable. Figure 5.25 shows how we can use randomly generated test cases to prove that each of the constraints is both consistent and falsifiable. Consistency requires that we can find a model that satisfies the constraint (l. 3 and 5–7). Falsifiability requires that we can find a model that violates the constraint (l. 11 and 13).

The above tests establish *local consistency*, so that each constraint can be satisfied on its own. This does not mean that your constraint system is satisfiable as a whole. To establish *global consistency* we need a global version of the function in lines 1–3 (Fig. 5.25) that checks not a given



M3
Ecore.ecore
«conformsTo»
M2
fsm abstract syntax (meta-model)
and
static well-formedness constraints for fsm
M1
coffeemachine.fsm/.xmi instance
M3
Ecore.ecore
M2
OCL abstract syntax (meta-model)
and
static well-formdeness constraints for OCL
M1
OCL Constraints (model)

Figure 5.26: *Two views on the language-conformance hierarchy for* `fsm` *with constraints and for OCL.* ***Left:*** *statically constraining instances of* `fsm`*.* ***Right:*** *OCL—as a language in the language-conformance hierarchy it is constrained itself*

constraint, but all our constraints on the same model—a recommended exercise. Normally, randomly satisfying many constraints might be difficult, but here we are using a generator of well-behaved models, so this will work well. On the other hand, we use the "free" generator for falsifiability tests (l. 10). Can you explain why?

Testing for consistency and validity are examples of a more general test strategy, *property-based testing*, which is well suited for testing language implementations. We demonstrate it more extensively in Sect. 6.7 by testing an implementation of a type checker.

Finally, neither scenario testing nor randomized testing of constraints suffices for quality assurance of static checkers of most languages. Typically, only when the back-end of the language tool is implemented unmet requirements on input models manifest themselves as problems with interpretation or code generation. If this happens, you need to add new regression scenarios to your collection and revise the constraints iteratively.

5.6 Constraints in the Language-Conformance Hierarchy

Constraints define model conformance and thus have a place in the hierarchy of models and meta-models defined in Sect. 3.9. The left part of Fig. 5.26 shows how constraints influence the conformance of `fsm` models: a valid model is restricted both by the meta-model (types, connections, and cardinalities) and by constraints specifying the restrictions that cannot be captured in the meta-model. Some constraints can be moved between diagrams and a textual constraint specification, for instance cardinalities can be expressed both ways (cf. p. 176). Language workbenches often store constraints within the meta-model; for example, Ecore allows this. Then the entire M2 layer is kept in a single file.

The right part of Fig. 5.26 changes the perspective: a constraint language (OCL) is also a DSL. In the new hierarchy, the constraints themselves are at level M1. From the OCL-as-a-language perspective these constraints are instances or models of OCL. The constraint language has its own abstract-syntax definition (the OCL meta-model at M2), and its own well-

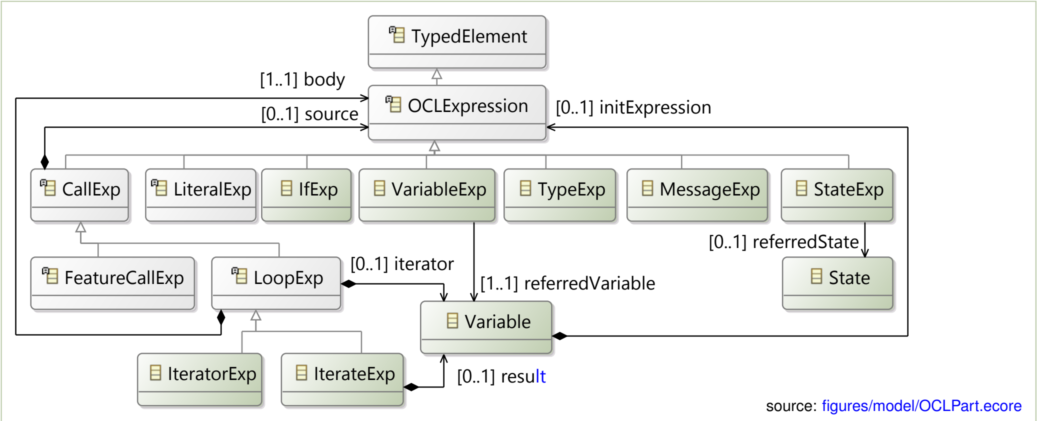


Figure 5.27: A view of the OCL expression meta-model in Ecore syntax, adapted from the OCL Specification, which uses UML [17]

```
context IterateExp
inv: self.result.initExpression->size() = 1
```

Figure 5.28: A result variable of an IterateExp must have an init expression (OCL)

formedness constraints (also M2). Yes! The OCL specification from OMG contains OCL constraints defining valid OCL constraints! [17]

Figure 5.27 shows a fragment of the meta-model from the specification. The fragment defines OCL expressions. An expression in OCL is typed, and it is either a call expression, a literal expression, a conditional expression, a variable expression, a type expression, a message expression, or a state expression. The figure details the variable expressions. Each variable expression has a reference to a variable object. The variable object itself may contain another expression as an initializer (`initExpression`).

The instances of this meta-model are further constrained. Figure 5.28 shows an example constraint, extracted from the OCL specification. Remarkably, these constraints do not talk about the elements in the `fsm` meta-model, but about the elements in the OCL meta-model, so they restrict not what state machines we can create, but *what constraints we can write*.

Exercise 5.19. The Xtext grammar specification language is also a DSL (Fig. 4.6). It has its own abstract syntax (meta-model) and its own static constraints. Study the code base of Xtext's implementation.[a] **a)** Identify the concrete-syntax definition (hint: Fig. 4.19), **b)** identify the abstract-syntax definition (the meta-model) (hint: Fig. 4.18), **c)** identify static validation constraints for the Xtext files.

Further Reading

Our formalization of class diagrams using logic was arguably a bit hasty. For example, we have not formalized the link between instances of a formula and the logical constraints. There are many research papers on this topic. We mention a few that we know first hand, as authors. A small and very concise definition can be

[a] https://github.com/eclipse/xtext-core

found in the work of Fahrenberg et al. [8], or in the definition of formal semantics for Clafer [14]. Semantically, Clafer can be seen as a simple class-diagramming language, with a few syntactic devices to keep the models concise. A graph-oriented perspective, as opposed to a logical view of the same problem of classes vs instances, is often found in categorical approaches, for instance in the work of Bak et al. [1].

The *transitive reduction* of a binary relation is a concept dual to the transitive closure. Instead of inferring new indirect binary connections in a graph, the transitive reduction removes them from a relation. Much less known than the closure, the transitive reduction is uniquely defined and well described in graph algorithms; see for example the book of Valiente [22]. It can be used to uncover the core dependence structure from a logical description, for example in model synthesis [20, 5].

The standard reference on OCL is the book of Warmer and Kleppe [23]. Chapter 3 presents guidelines for writing constraints (see esp. Section 3.10 on tips and tricks). A more concise, but still comprehensive, presentation can be found in a 30-page-long tutorial paper of Cabot and Gogolla [2]. Another tutorial-like resource is the slides of a course on OCL by Demuth [7]. The current official OCL specification can always be found at http://www.omg.org/spec/OCL/ (last checked 2022/09). The specification is not particularly useful for learning OCL. It serves as a reference definition. Chapter 7 (The OCL Language Description) is certainly worth looking into.

The most concise overview text about Alloy is Jackson's recent ACM Communications article [10]. His book [12] and the journal paper on Alloy [11] contain short and interesting critiques of OCL. He contrasts OCL with Alloy, emphasizing the difference between the relational and logical styles. Besides Alloy, other modeling languages take constraints to the extreme, most notably FORMULA [13] and Clafer [1]. The main distinguishing feature of FORMULA is the semantics based on SMT solving, which allows first-class treatment of numbers (instead of just simple entities like in Alloy). The main advantage of Clafer is its concise economical syntax exploiting the strengths of feature modeling. Unlike Alloy, Clafer allows part-of relationships to be directly specified, like in class diagrams.

Additional Exercises

Exercise 5.20. Which of the following first-order sentences hold for the generalization hierarchy shown in Fig. 5.29?

a) $\forall x.\, \mathsf{HybridEngine}(x) \rightarrow \mathsf{CombustionEngine}(x) \wedge \mathsf{ElectricMotor}(x)$?

b) $\forall x.\, \mathsf{DieselEngine}(x) \rightarrow \mathsf{ElectricMotor}(x)$?

c) $\forall x.\, \mathsf{DieselEngine}(x) \rightarrow \mathsf{CombustionEngine}(x) \wedge \mathsf{ElectricMotor}(x)$?

d) $\forall x.\, \mathsf{CombustionEngine}(x) \wedge \mathsf{HybridEngine}(x) \rightarrow \mathsf{ElectricMotor}(x)$?

Exercise 5.21. Recall the OCL higher-order function `isUnique` (see Tbl. 5.4). Does the GPL you use for DSL implementations offer this function in the library? If yes, what is its name and type? Are there differences from OCL? If no, implement the function yourself and use it later to solve Exercise 5.26c.

Exercise 5.22. A common pattern of binary universal quantification is *all-different*: the quantifier enforcing a property for *all pairs of different elements* in a collection. Both Constraint **C1** and **C2** are of this form; see also Fig. 5.16. Implement `forAllDifferent` in a language of your choice and use it to simplify **C1** by

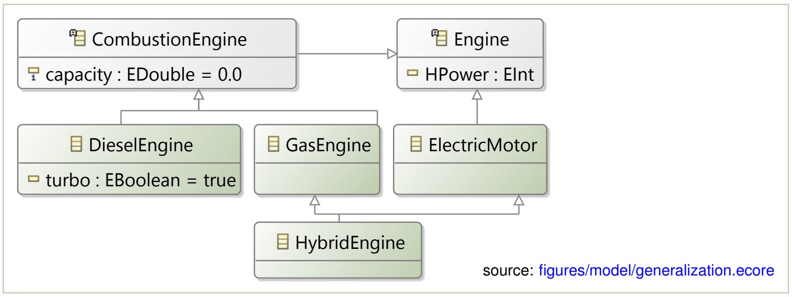


Figure 5.29: An example generalization hierarchy of car engine designs

eliminating the precondition and implication. We show Scala implementations in scala/src/main/scala/dsldesign/scala/emf.scala as an example.

Exercise 5.23. Recall the `closure` operation from OCL that computes the reflexive transitive closure of a binary relation specified as a lambda expression. In Scala the type of the operation would be approximately the following:

```
def closure[A] (self: Seq[A]) (R: A => Seq[A]): Seq[A]
```

Implement this operation in Scala or in another GPL. Refactor the implementation of Constraint **C5** in Tbl. 5.2 to use the new operation.

Exercise 5.24. Derived attributes can be implemented using extension methods in many GPLs, including Scala, Kotlin, Groovy, Xtend, C#, and F#. Implement `isInitial` (Fig. 5.11) as an extension method in a GPL of your choice. Consider using value caching (for instance `lazy val` in Scala), to compute the derived value only at the first access and retrieve it from a cache all the subsequent times.

Exercise 5.25. Recall the class types in the diagram of Fig. 5.1 on p. 143. Write the following commutativity constraints for this diagram (in logic, in a programming language, or in a dedicated constraint language):

a) Each person is listed in the set of parents of each of its children.
b) Each person should be included in the set of children of its own parents.

Exercise 5.26. Consider a simple meta-model of a language for describing organization of trips (Fig. 5.30). Write the following constraints for this meta-model:

a) The vehicles associated with a trip need to be large enough to accommodate all the involved passengers: for each trip, the number of passengers must be smaller than or equal to the number of seats in the involved vehicles.
b) For each vehicle participating in a trip, a driver is included in this trip that drives this vehicle, and this person is also in the list of passengers of this trip.
c) Every car is uniquely identified by its registration plate. Write the constraint first in the context of the `Trip` class, then in the context of the `TripModel` class, avoiding use of `allInstances` (if using OCL).

Exercise 5.27. Consider the naive meta-model in Fig. 5.31 describing a car.

a) For this meta-model, write the following constraint: *Driver's seat in a car must be a seat in the same car.* Use a formalism of your choice.

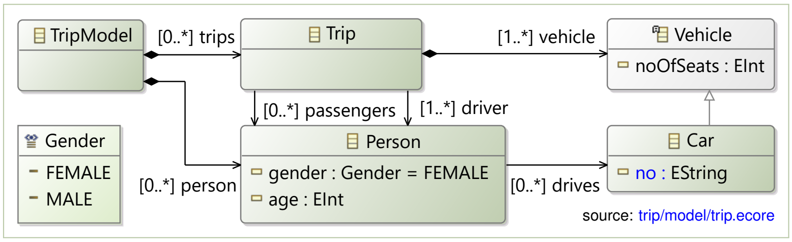


Figure 5.30: *A trip meta-model: describing simple travel arrangements*

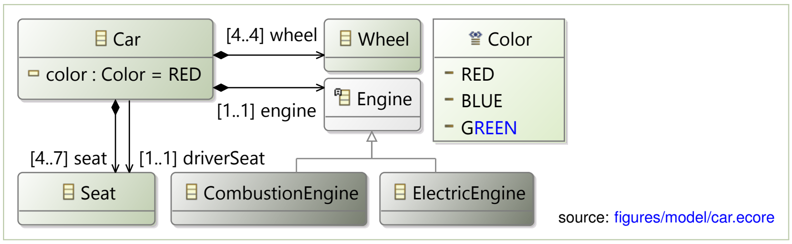


Figure 5.31: *A simple model of structural components of a passenger car*

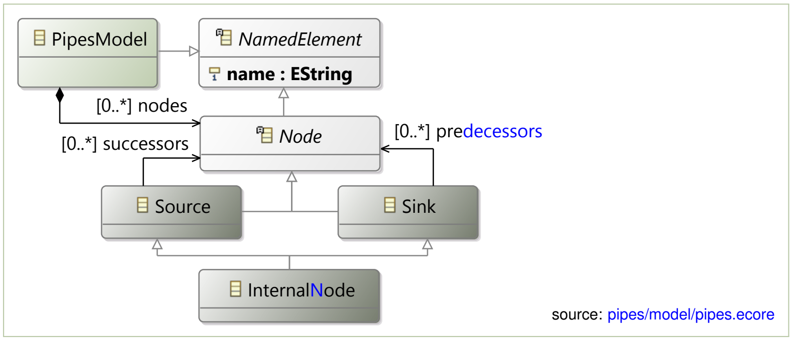


Figure 5.32: *A meta-model for the core of a flow-based web composition language*

b) Refactor the meta-model to make `driverSeat` an attribute in the `Seat` class (an attribute). How can you enforce the above constraint now?

c) Refactor the model to split the passenger seats (`4..7`) and the driver seat into two separate containments, `passengerSeats` and `driverSeat`. How can you now access the set of all seats? Implement a derived attribute `seats` that is the union of values of the two new attributes. Use derived properties if writing in OCL, extension methods, protected code blocks in Java (in code generated by Ecore), or just a static function in other languages.

Exercise 5.28. Figure 5.32 presents a meta-model of a flow language for creating web mash-ups.[12] A model consists of nodes, which are further divided into sources (where the flow starts), internal nodes (processing nodes), and sinks (rendering nodes, where the flow ends). **a)** Is the instance shown in Fig. 5.33 a valid instance of this meta-model? **b)** Figure 5.34 shows a constraint for this meta-model. Does the instance satisfy this constraint? If yes, argue how each constraint is satisfied. If no, indicate the violating model elements.

[12] One such language was Yahoo! Pipes, now defunct, cf. https://en.wikipedia.org/wiki/Yahoo!_Pipes. Other services use similar languages, for instance: http://www.pipes.digital.

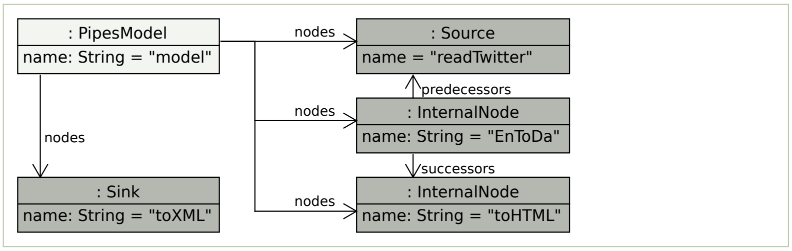


Figure 5.33: An example instance for the meta-model of Fig. 5.32

```
// Xtend
def boolean constraint (PipesModel it) {
  nodes.filter [it | it instanceof Source].size == 1
  && nodes.filter [it | it instanceof Sink].size == 1
}
```

```
// Scala
(m: PipesModel) =>
  m.getNodes.filter { _.isInstanceOf[Source] }.size == 1 &&
  m.getNodes.filter { _.isInstanceOf[Sink] }.size == 1
```

source: pipes.scala/src/main/scala/dsldesign/pipes/scala/constraints.scala

Figure 5.34: A class cardinality constraint for Pipes in Xtend and Scala

Exercise 5.29. Constrain the meta-model of Fig. 5.32 so that from each `Source` instance one can reach a `Sink` instance (not just an `InternalNode`) via a series of `successors` links. Warning: depending on your constraint language this may require implementing a depth-first search in the graph (if transitive closure is not supported explicitly).

Exercise 5.30. For each of the following constraints indicate the *preferred context class* in the Pipes meta-model of Fig. 5.32.

a) Every source has exactly one successor

b) An internal node has at least two successors or its name is an empty string

c) There is exactly one node whose name is “abrakadabra”

Exercise 5.31. For the model of Figure 5.1 write the constraint that if person `A` is a parent of `B` then the two persons are distinct. Test the constraint on a negative example (a violating instance) of your design.

Exercise 5.32. This and several following exercises use the same running example of a printing infrastructure in an office. The first meta-model is shown in Fig. 5.35.[13] (Some constraints are shown in the project printers.scala in the book code repository.) Write the following constraints for this meta-model:

a) *Every printer pool that has a fax, also has a printer.* Write the constraint in the context of the `PrinterPool` class. Create an instance that satisfies the constraint and verify by running the validation to see whether this is indeed the case. Create an instance that violates it and verify that this is indeed the case.

b) Write the constraint from the previous point in the context of class `Fax`. Use the same positive and negative instances to test it. Reflect on the differences between the two formulations.

[13]These exercises are inspired by the submission for the standardization process of *Common Variability Language* within the Object Management Group [21]. This part of the submission has been prepared by Krzysztof Czarnecki, Kacper Bąk, and Andrzej Wąsowski.

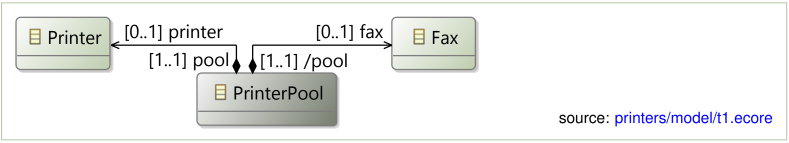


Figure 5.35: *An example printer pool model* `t1`

Exercise 5.33. Write the following constraint in the context of the printer pool class in the meta-model of Fig. 5.36: *Each printer pool with a fax must have a printer, and each printer pool with a copier must have a scanner and a printer*. Create two instances, one that satisfies and one that violates the constraint. Test whether this is indeed the case, by evaluating the constraint on the instances.

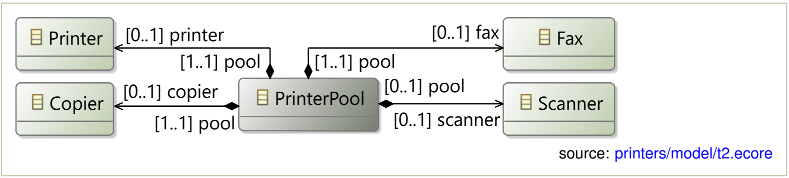


Figure 5.36: *A class diagram* `t2` *showing a printer pool with scanners and copiers*

Exercise 5.34. For the meta-model `t3` in Fig. 5.37, write the following constraint: *PrinterPool's minimum speed must be 300 units lower than its regular speed*. Validate the constraint with a positive and a negative test instance.

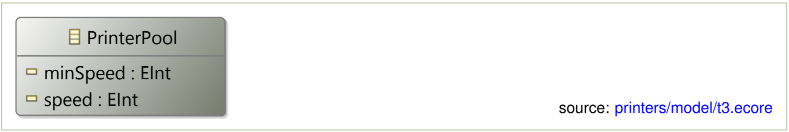


Figure 5.37: *A class diagram of the* `t3` *meta-model with attributes*

Exercise 5.35. For the meta-model `t4` in Fig. 5.38, write the following constraint in the context of the class `Printer`: *Every color printer has a colorPrinterHead*. Validate the constraint with a positive and a negative instance.

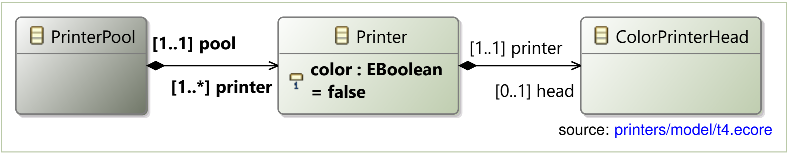


Figure 5.38: *A meta-model* `t4` *which allows that printers have color printing heads*

Exercise 5.36. For the meta-model `t5` (Fig. 5.39) formulate a constraint that *a color-capable printer pool contains at least one color-capable printer* (in the context of class `Printer`). Validate it using a positive and a negative instance.

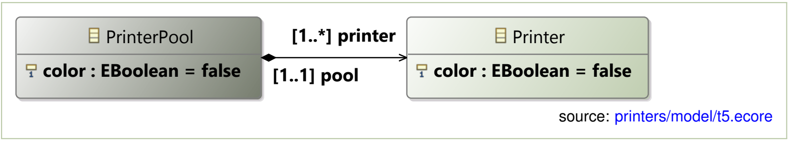


Figure 5.39: *Meta-model* `t5` *with color printer pools and color printers*

Exercise 5.37. For the meta-model `t6` of Fig. 5.40 Write the following constraints: **a)** *If a printer pool contains a color scanner, then all its printers must be color printers,* **b)** *If a printer pool contains a color scanner, then it must contain a color printer.* Write the constraints in the context of the class `PrinterPool`, and test them using negative and positive instances.

Now write both of these constraints first in the context of `Scanner` and then in the context of `Printer`, so four new constraints in total. The last case is particularly unwieldy. Use the same instances to test the constraints in the new contexts. Discuss the differences that context changes introduce to constraints.

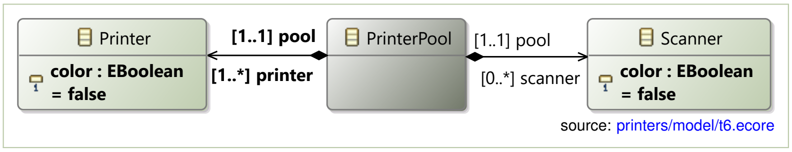


Figure 5.40: Colored scanners and printers in meta-model `t6`

Exercise 5.38. For the diagram of Fig. 5.40 assert that *there is at most one color printer in any pool*. Test the constraint on a positive and a negative instance.

Exercise 5.39. Figure 5.41 presents a simplified meta-model for SQL queries.

a) Write a constraint that every `SelectQuery` selects from exactly one table, and all columns come from the same table. Write it in the context of `SelectQuery`.

b) Write the same constraint in the context of the `Model` class.

c) Forget the above. Now a query can draw from several tables, but the tables used in a query must not have columns with the same names. Write this constraint in the context of `SelectQuery`.

d) Now combine both ideas. Write a constraint that all column names accessed in a single query are unambiguous: in each query, if a column is accessed, no other column in the referred tables has the same name. Context: `SelectQuery`.

If you do the exercise in Scala, an empty file for constraints is prepared at sql.scala/src/main/scala/dsldesign/sql/scala/constraints.scala.

Exercise 5.40. In the meta-model of Fig. 3.20 (p. 81) sub-features are contained in the `subfeatures` collection of the parent feature. If features are part of a group, an object of type `Group1` is placed under the feature object with references to the features that are group members.

a) Write a constraint enforcing that a group can only contain sub-features of its parent, and not of other features. Figure 5.42 shows a positive and a negative instance. The latter should be prevented by the constraint.

b) Write a constraint stating that any two groups nested under the same feature cannot overlap (they have disjoint sets of members).

Exercise 5.41. Write the following constraints over the instances of the Pascal's triangle meta-model of Fig. 3.18 on p. 80:

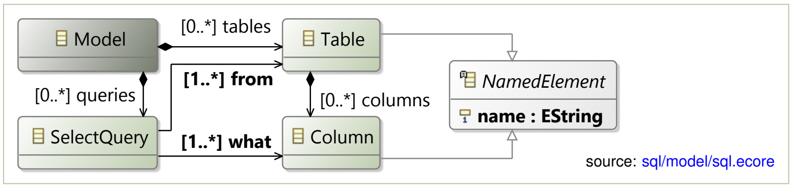


Figure 5.41: *A meta-model for a very simple core of select queries in SQL*

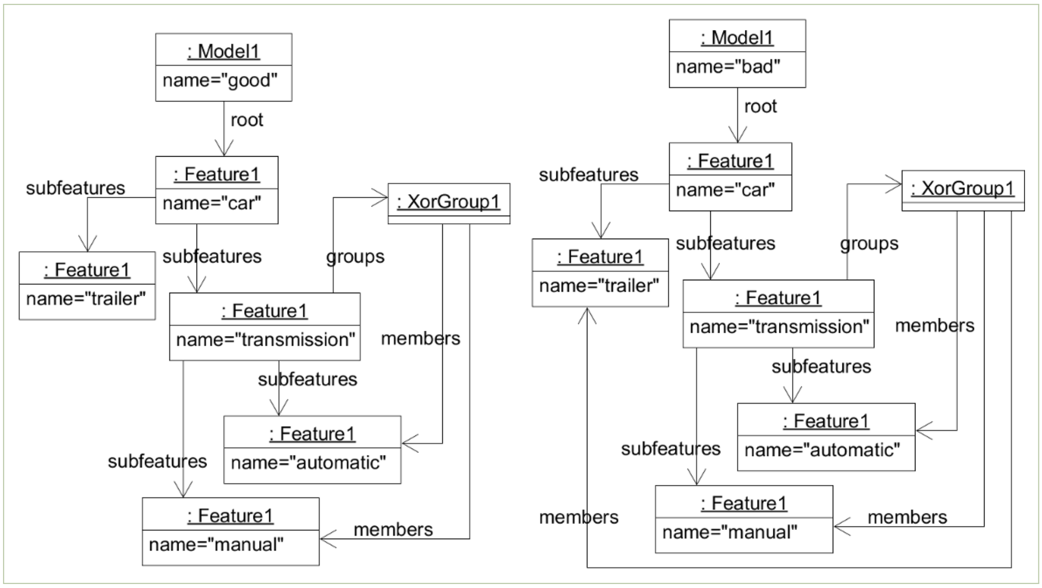


Figure 5.42: *A good (left) and a bad (right) instance for Exercise 5.40 question* ***a)****. The right one should be prevented by constraints*

a) The value of each internal entry is equal to the sum of the parent values (internal entries are defined as the entries that have two parents).

b) For every row n, the parents of all the nodes in the row are at row $n-1$.

Exercise 5.42. Figure 5.43 shows a simple meta-model for relational schemas. Its instances store primary keys in the `primaryKeys` collection, and foreign keys in the `refersTo` collection. Write a constraint enforcing that a primary key column cannot also be a foreign key and vice versa. Note that a column is a primary key and a foreign key at the same time if it is both pointed to from a table, and itself refers to a table. Test the constraint on a positive and a negative instance.

Exercise 5.43. We would like to constrain instances of the pipes meta-model (Fig. 5.32) so that each instance has at most one source, at least one sink, and at least one internal node using the following constraint (Xtend):

```
def boolean constraint (PipesModel it) {
  nodes.exists [it | it instanceof Source]
  || nodes.exists [it | it instanceof Sink]
  || nodes.exists [it | it instanceof InternalNode]
}
```

Testing shows that this constraint is incorrect. Propose an improvement, in a chosen constraint language, and validate it with a positive and a negative instance.

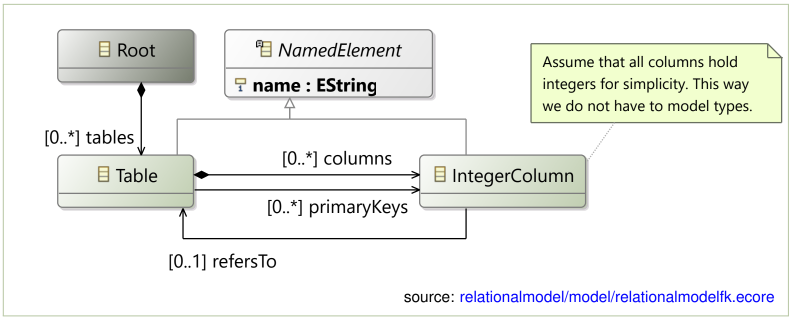

Root
NamedElement
name : EString
Assume that all columns hold integers for simplicity. This way we do not have to model types.
[0..*] tables
Table
[0..*] columns
IntegerColumn
[0..*] primaryKeys
[0..1] refersTo
source: relationalmodel/model/relationalmodelfk.ecore

Figure 5.43: A simple meta-model for relational schemas with tables and integer columns, primary keys, and foreign keys

Exercise 5.44. We would like to constrain the instances of the pipes meta-model (Fig. 5.32), so that each sink has at least one predecessor. We do this using the following constraint written in the context of class `Sink` (Xtend):

```
def boolean constraint (Sink it) { ! predecessors.empty }
```

Rewrite this constraint in the context of the `PipesModel` class in a constraint language of your choice.

Exercise 5.45. The pipes meta-model in Fig. 5.32 contains a flaw. It allows sinks to be predecessors (while sinks should have no outgoing edges), and sources to be successors (while sources should have no incoming edges). Fix this by changing the meta-model. For the repaired meta-model write a constraint enforcing that successors and predecessors are opposite associations, i.e.,, if a node a is a predecessor of a node b then b is a successor of the node a, and vice versa (so formalize in a constraint language the EMF `EOpposite` mechanism).

Exercise 5.46. We want to restrict the Pipes meta-model of Fig. 5.32 with the two following constraints (enforced for all sinks and all sources, written in Xtend):

```
def boolean constraint (Sink it) { !predecessors.empty }
def boolean constraint (Source it) { !successors.empty }
```

Alternatively these constraints could be incorporated into the meta-model. Revise the meta-model to contain these constraints directly in the class diagram.

Exercise 5.47. We want to constraint the Pipes meta-model of Fig. 5.32 so that each instance model has exactly one source and exactly one sink:

```
def boolean constraint (PipesModel it) {
  nodes.exists [it | it instanceof Source] &&
  nodes.exists [it | it instanceof Sink]
}
```

Is this implementation correct? If so, explain why. If not, specify an example instance on which the English specification and the Xtend constraint differ. To test this, you may need to translate the constraint into the constraint language used in your modeling environment.

Exercise 5.48. Recall the constraint presented in Fig. 5.34 for the model of Fig. 5.32. What are the suitable test cases for testing the constraint? Discuss how you would select test cases for this constraint (including example test cases).

Exercise 5.49. Formulate the constraints from Exercise 5.37 in Java, C#, or Python (in the context of the class `PrinterPool`), without using anonymous and higher-order functions. Comment on the difference in writing constraints using functional (declarative) and imperative style. Which version of the constraint is more readable? Why?

Exercise 5.50. Recall the key fragment of the Ecore meta-model presented in Fig. 3.25 (p. 83). Write a constraint that restricts this model's instances to only allow generalization of `EClasses` (`eSuperTypes`) by other `EClass` instances in the same `EPackage`. It should not be allowed to generalize `EClasses` across package boundaries. For simplicity, do not use the full Ecore meta-model, just the one presented in Fig. 3.25.

Exercise 5.51. The micro Ecore meta-model of Fig. 3.25 on p. 83 allows that an `EClass` is a super-type of itself. Write a constraint that disallows that. Only disallow direct (non-transitive) generalization of an `EClass` by itself.

Now, strengthen the constraint to also disallow an `EClass` to be an *indirect* (transitive) generalization of itself. You can use the provided helper function `allSuperTypes` formulated in Xtend:

```
// Get the set of all super-types of c, including c and classes in r
def static private Set<EClass> allSuperTypes(EClass c, Set<EClass> r) {
  if (r.contains(c)) return r
  r.add (c)
  c.ESuperTypes.toSet.fold(r, [r2, t| allSuperTypes(t, r2)])
}
```

Exercise 5.52. Recreate the model of Fig. 5.31 in Alloy. Use Alloy Analyzer to count how many possible configurations are possible. Then add the constraint from Exercise 5.27a and repeat the count. Reflect on this result. Has the number changed? Why?

References

[1] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. "Clafer: Unifying class and feature modeling". In: *Software and System Modeling* 15.3 (2016) (cit. p. 190).

[2] Jordi Cabot and Martin Gogolla. "Object constraint language (OCL): A definitive guide". In: *12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-Driven Engineering*. SFM. 2012 (cit. p. 190).

[3] Koen Claessen and John Hughes. "QuickCheck: A lightweight tool for random testing of Haskell programs". In: *International Conference on Functional Programming*. ICFP. 2000 (cit. p. 185).

[4] E. F. Codd. "Relational completeness of data base sublanguages". In: *Research Report / RJ / IBM / San Jose, California* RJ987 (1972) (cit. p. 162).

[5] Krzysztof Czarnecki and Andrzej Wąsowski. "Feature diagrams and logics: there and back again". In: *Software Product Line Conference*. SPLC. 2007 (cit. p. 190).

[6] Rina Dechter. *Constraint Processing*. Morgan-Kauffman, 2003 (cit. p. 175).

[7] Birgit Demuth. *OCL (Object Constraint Language) by Example*. http://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf. 2009 (cit. p. 190).

[8] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. “Sound merging and differencing for class diagrams”. In: *Fundamental Approaches to Software Engineering (FASE)*. 2014 (cit. p. 190).

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995 (cit. p. 168).

[10] Daniel Jackson. “Alloy: A language and tool for exploring software designs”. In: *Commun. ACM* 62.9 (2019) (cit. p. 190).

[11] Daniel Jackson. “Alloy: A lightweight object modelling notation”. In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (2002), pp. 256–290 (cit. p. 190).

[12] Daniel Jackson. *Software Abstractions*. MIT Press, 2006 (cit. pp. 167, 172, 190).

[13] Ethan K. Jackson and Wolfram Schulte. “FORMULA 2.0: A language for formal specifications”. In: *Unifying Theories of Programming and Formal Engineering Methods - International Training School on Soft. Eng., Held at ICTAC’13*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8050. Lecture Notes in Computer Science. Springer, 2013 (cit. p. 190).

[14] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. “Clafer: Lightweight modeling of structure, behaviour, and variability”. In: *Art Sci. Eng. Program.* 3.1 (2019), p. 2 (cit. p. 190).

[15] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018 (cit. p. 174).

[16] Stanisław Lem. *The Cyberiad: Fables for the Cybernetic Age*. Trans. by Michael Kandel. The original from 1965 in Polish. Harcourt Brace, 1974 (cit. p. 143).

[17] Object Management Group. *OCL Specification version 2.2*. http://www.omg.org/spec/OCL/2.2/. 2010 (cit. pp. 167, 189).

[18] Daniel Ratiu, Markus Völter, and Domenik Pavletic. “Automated testing of DSL implementations—experiences from building mbeddr”. In: *Software Quality Journal* 26.4 (2018), pp. 1483–1518 (cit. p. 181).

[19] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006 (cit. p. 175).

[20] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wąsowski, and Krzysztof Czarnecki. “Efficient synthesis of feature models”. In: *Information and Software Technology* 56.9 (2014) (cit. p. 190).

[21] Submitters and supporters. *Common Variability Language. OMG Revised Submission*. 2012 (cit. p. 193).

[22] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer, 2002 (cit. p. 190).

[23] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 2003 (cit. p. 190).

You won't find a lemon
in the vegetable container.
the spouse to one of the authors

6 Static Semantics with Type Systems

Type systems are a common complement to structural constraints in enforcing static semantics on a program text, and are particularly useful if you need to track recursive properties on inductive syntax types (meta-models with cycles over containment relations). In this chapter, our goal is to explain what types and type systems are, to show how to build a simple one, and to discuss when it is practical to use a type system instead of structural constraints.

Types are labels decorating an abstract-syntax tree with limited information about the meaning (semantics) of the individual syntax nodes. A *type checker* does two things simultaneously: (i) it infers the decorations summarizing non-local properties in a syntax tree, and (ii) it enforces structural constraints on the inferred decorations. This effectively constrains elements and properties placed arbitrarily far from each other in the syntax graph.

Type systems are particularly useful when types are not a direct property of a syntax object, but rather emerge from properties of an entire sub-tree of syntax objects. Thus type systems are a natural generalization of structural constraints. They add a step of additional information inference before enforcing structural constraints on the inferred labels. Just like in Sect. 5.2, in type systems we tend to use constraints that are directly executable (unlike the constraints in Alloy, which need to be solved). Consequently, executable structural constraints, presented in the previous chapter, are the simplest possible type system—one which infers no additional properties beyond what is found directly in the syntax.

Example 16. `Prpro` is an example language loosely inspired by the probabilistic programming framework PyMC.[1] PyMC's interface can be seen as an *internal domain-specific language* (Chapter 10) for describing Bayesian probabilistic models. In contrast, `prpro`, developed partly in this chapter, is an external DSL but with similar goals.

In the first example model in `prpro`, we declare two named constants, x and y, followed by a normal distribution with the mean parameter μ equal to $x+z$ and the standard deviation $\sigma = y$.

$$\begin{aligned} &x = 0\\ &y = 0\\ &u \sim \mathcal{N}(\mu = x + z, \sigma = y) \end{aligned}$$

A type checker for `prpro` should flag an error above: the name z is used, but undeclared. If z was declared, but had an incompatible type instead (say a string of characters), an error should be raised as well. Importantly, discovering the type of z may require bringing information from far away. The variable z could have been declared very far away in a large model, with many other declarations placed before the use. In a complex language, it is typically impossible to write a static navigation expression over the abstract-syntax tree that finds the declaration of z and constrains it to exist, exactly because of this unbounded distance. A type system must perform work that resembles a transitive closure: traversing and collecting information from the entire model.

In probabilistic programming, parameters of a distribution do not have to be fixed values. In the example below, x is itself governed by a, so-called, prior distribution.[2] We do not know what is the value of x but we do know that it is a floating-point number selected from the interval $(-1;1)$ with a uniform probability density:

$$\begin{aligned}
&x \sim \mathscr{U}(-1,1)\\
&y = 1\\
&u \sim \mathscr{N}(\mu = x+y, \sigma = y)
\end{aligned}$$

This flexibility influences how we think about types in `prpro`. It turns out that μ does not have to be a floating-point number, but can also be a probability distribution over numbers. In the above model, not only x is a distribution, but also $x+y$ and μ. The expression $x+y$ represents the distribution of x shifted by a constant y. This also means that the normal distribution, in the last line, is not a pure normal distribution, but a distribution that arises from averaging normal distributions with means (μ) selected from the distribution $x+y$. To perform reasoning of this kind, we need to traverse the entire slice of the model that is involved in calculating μ, including declarations of all involved variables—a perfect task for a type checker.

We assume the following definition of a type system after Pierce [10].

Definition 6.1. *A type system is a tractable syntactic method for proving the absence of selected errors in the construction of a model (program) by classifying syntax elements according to their relevant properties.*

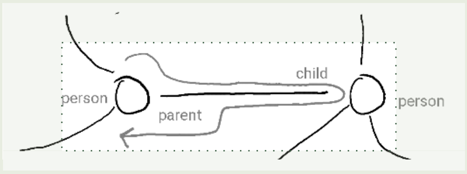
This abstract definition calls for a few explanations. First, a type system shall be *tractable*. The algorithm for establishing type correctness should be efficient, typically polynomial in the size of the input model. This is why we want to use only executable constraints. Theoretically, we could encode type correctness as constraints with free variables in a sufficiently rich logic, but solving them would be very hard; far from polynomial time. Type checkers typically use algorithms that infer types *inductively* by traversing the syntax tree (see the info box on p. 203).

A type checker is not a universal verification tool. It is constructed to prove limited concrete properties, to detect *selected errors*. These could be errors in computation (like memory safety), but could also be errors in how

[1] https://docs.pymc.io/, accessed 2022/09

[2] Do not worry about Bayesian models, priors, density functions, etc. if you do not know them. They do not have major importance in the rest of the book.

What Are Inductive Properties?



Exploring a local fragment of a model instance to check a parent-child property.

In Chapter 5, we focused on requirements (restrictions) that could be expressed directly in terms of meta-model types, through a localized inspection. For example: *A person object should be included among its child object's parents*. This property is directly expressible as a computation over a fixed number of objects, without navigating arbitrarily far from the context object in the syntax graph.

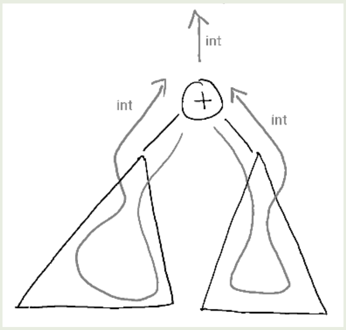
In contrast, the type of an arithmetic expression is not directly computable by just examining the instance object in question (an expression node) and a small number of its neighbors. Consider the rule for typing a binary addition expression: *The result of binary addition is an integer if both of its arguments are integers*. This rule, similarly to the other constraints we considered before, can be split into two parts:

Premises (inductive):	Both sub-expressions evaluate to an integer number
Conclusion (structural):	The result of binary addition is an integer

To establish that the conclusion holds, we first need to establish the premises. Often, and also in this case, enforcing the premises may require exploration of an arbitrary large abstract-syntax tree, using the same rule applied to a smaller part of the model. What if the left operand is an addition expression itself? And what if the left operand of the left operand is an addition again? You can see that we might be dealing with an arbitrary large sub-tree whose type is not directly known. We shall apply the same typing rule to smaller and smaller sub-expressions, until we hit the leaves (constants and variable references), where we can decide with certainty that their type is integer, without invoking the rule recursively.

An *inductive property* is a property which requires multiple recursive checks of itself on decreasing pieces of syntax. In language implementation, we usually encounter mutually recursive inductive properties—sets of rules that recursively invoke each other. Establishing that they hold requires exploring arbitrary large parts of the instance. This resembles reflexive transitive closure. Indeed, transitive closure is an example of an inductive property.

We use *structural induction*, which differs from mathematical induction you may recall from high school. Mathematical induction derives facts for natural numbers if they hold for smaller numbers. Structural induction establishes that a property holds for a syntax tree if it holds for smaller sub-trees. The process terminates, since at every inductive step we are considering smaller trees, until we arrive at basic terms, which can be typed non-inductively (without further recursion).



Establishing an inductive property may require exploration of arbitrary large sub-trees of an AST.

the model instance is structured—most useful for non-behavioral languages. For instance, for a modeling language of electrical circuits we could imagine a type system which ensures that alternating current (AC) is not connected to direct current (DC) ports, or two AC ports with the wrong voltage.

Type checking is a *syntactic method* that operates directly on the syntax tree, or an instance of a meta-model, without building complex and

```scala
val example1: Model = List (
  Data ("x", VectorTy(17, PosFloatTy)),
  Data ("y", VectorTy(17, FloatTy)),

  Let ("Beta0", Uniform (CstI (-200), CstI(200))),
  Let ("Beta1", Uniform (CstI (-200), CstI(200))),
  Let ("sigma", Uniform (CstF (0), CstI (100)) ),

  Let ("y",
    Normal(
      mu = BExpr (
        BExpr (VarRef ("Beta1"), Mult, VarRef ("x")),
        Plus,
        VarRef ("Beta0")),
     sigma = VarRef ("sigma") )
  )
)
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/testCases.scala

Figure 6.1: *An example* `prpro` *model in abstract syntax. We build a type checker for such models in this syntax*

expensive representations. It works by *classifying syntax elements*, labeling them with discrete information representing a property, for instance *"this expression will produce an integer value,"* or *"this wire carries DC current."*

Type checking is most used for algebraic DSLs and languages with expressions. In expression languages, a model fragment is built from hierarchies of atoms and operators, and then an inductive definition is natural: we can locally reason about the types of larger expressions based on the types of smaller expressions. In contrast, the structural first-order constraints discussed in Chapter 5 are best suited for properties that are local, and related to types of direct connections in the abstract syntax. If a property is natural to write as an executable constraint over abstract syntax without types then avoid constructing a type system altogether.

In this chapter, we develop and reflect upon the basics of a type system for `prpro`, our small Bayesian modeling language. Once you have an abstract syntax for your language, there are three parts of a type system that need to be developed: (i) the language of types, (ii) the typing hierarchy, for languages that need sub-typing, and (iii) the type-checking algorithm. We go through all of them in order below, using examples in Scala and Java with Ecore. The examples are easy to recast in any other modern GPL.

6.1 Abstract Syntax

We develop the running example in two styles: object-oriented (using Ecore and Java) and functional (using Scala and its algebraic data types). Figure 6.1 shows a simple `prpro` model example in the abstract syntax in Scala, itself defined in Fig. 6.2. Figure 6.3 shows the corresponding meta-model in Ecore. Both definitions use the same type names and relations. A `Model` is an ordered list of `Declarations` binding values to names (l. 31). A declaration is an abstract type, with two concrete realizations (l. 8). Either we declare (`Let`) a binding of a name to an expression value, or we declare a named data set (`Data`). To keep the example small, `prpro` lacks a mechanism

```scala
abstract trait NamedElement:
  val name: String

trait Typeable:
  def getTy: Ty
  private def setTy (ty: Ty): Ty

enum Declaration extends NamedElement:
  case Let (name: String, value: Expression)
  case Data (name: String, ty: VectorTy)

sealed abstract trait Expression extends Typeable

enum Distribution extends Expression:
  case Uniform (lo: Expression, hi: Expression)
  case Normal (mu: Expression, sigma: Expression)

final case class BExpr (
  left: Expression,
  operator: Operator,
  right: Expression
) extends Expression

final case class VarRef (name: String) extends NamedElement, Expression
final case class CstI (value: Int) extends Expression
final case class CstF (value: Double) extends Expression

enum Operator:
  case Plus, Minus, Mult, Div

type Model = List[Declaration]
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt.scala

Figure 6.2: Algebraic data types representing the abstract syntax of `prpro`*, a simple probabilistic modeling language. See also Fig. 6.3*

to acquire the data, such as a URI or a path to a file. We only specify the type. `Expressions`, used in let-bindings, are divided into: `Distribution` expressions, binary expressions (`BExpr`), and simple expressions. In `prpro` we can combine distributions: so we can use distributions as elements in expressions. For example, we can compute a sum of distributions, or use distributions as values for parameters of other distributions (so-called prior distributions). The simple expressions are constant literals (both integer and floating point) and variable references, which refer to the expression or data set bound to a name. We expect that a variable is bound before it is referenced.

6.2 The Language of Types

Types are typically defined inductively. This means that simple values are described by simple types, complex values have complex types, and complex types are created from simpler types. Thus types are themselves expressions! Indeed, they are instances of another language—the *type language*. A type language is an abstraction of the language we apply types to, the *typed language*, here `prpro`. It follows the same core structure, but leaves out many details inessential for the property tracked by the

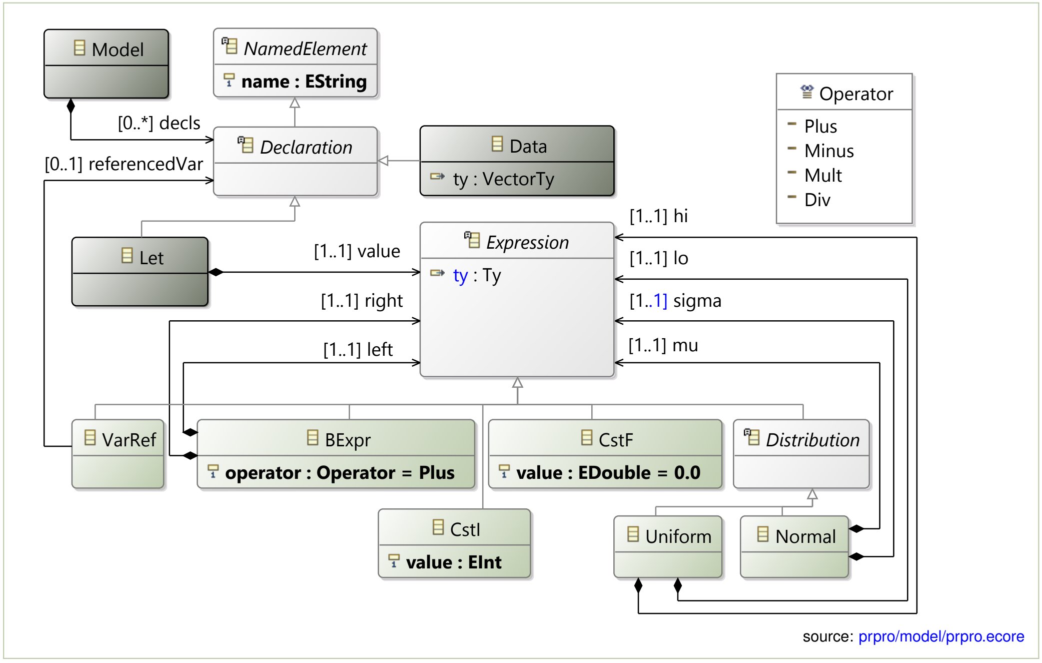


Figure 6.3: *An Ecore meta-model for* `prpro` *that follows a similar design to the ADTs in Fig. 6.2*

type system. The process of typing is the process of abstracting model syntax elements by the type language elements. Designing a type language amounts to a systematic inspection of the meta-classes of the typed language, asking what should be inferred about them, and what could be left out.

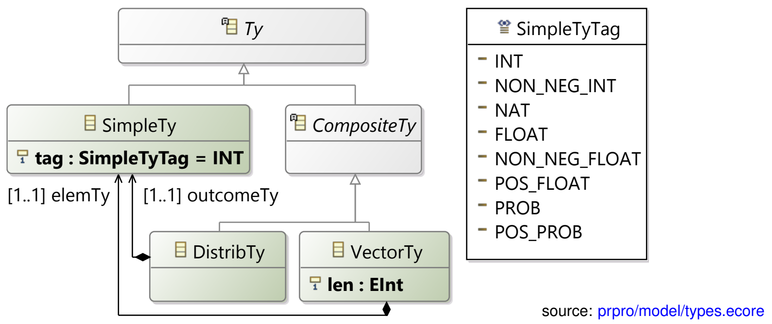
Since types are a language, we can specify their syntax using a meta-model or algebraic data types. Figures 6.4 and 6.5 show the abstract syntax for `prpro` types. `Prpro` has integer and floating-point values. In a probabilistic modeling language, we may want to control the ranges of numeric values and constants, so that we can distinguish distributions that generate positive values only, or parameters (like standard deviation) which only take non-negative values. Also, we might want to pay special attention to numbers between zero and one (probabilities), and whether a probability of zero is allowed for a given expression location. This leads to the following set of simple numeric types: integers, non-negative integers, naturals, floats, non-negative floats, probabilities, and positive probabilities. In Fig. 6.4, these are defined in lines 3–5; in Fig. 6.5 in the enumeration SimpleTyTag.

Composite types are defined as enumeration cases in Scala (lines 7–9 in Fig. 6.4) and as concrete classes in Ecore (the bottom part of the diagram in Fig. 6.5). `Prpro` includes binary expressions, distribution expressions, and declarations of named variables bound to distributions and data sets. What types of values can arise from these constructs? Binary expressions will have types arising from the combined sub-expressions. Obviously, if the sub-expressions have simple types (for instance they are integer constants), then

```scala
sealed abstract trait Ty

enum SimpleTy extends Ty:
  case IntTy, NonNegIntTy, NatTy, FloatTy,
    NonNegFloatTy, PosFloatTy, ProbTy, PosProbTy

enum CompositeTy extends Ty:
  case VectorTy (len: Int, elemTy: SimpleTy)
  case DistribTy (outcomeTy: SimpleTy)
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/types.scala

Figure 6.4: An ADT representing a type language for `prpro`*, a simple probabilistic modeling language. See also an Ecore definition of the type language in Fig. 6.5*



source: prpro/model/types.ecore

Figure 6.5: An Ecore meta-model for the language of types for `prpro` *that follows a similar design to the ADTs in Fig. 6.4*

the expression can inherit a simple type. What if we combine two distribution expressions? We shall obtain a distribution! What if we refer to a named data set? It is convenient to interpret data sets as vectors (`VectorTy`) of data elements. We want to track how long the vectors are, and what is the type of elements. For simplicity, we only admit vectors of simple types. In the developments below, we assume that a vector's length is greater than one.

A `prpro` executor needs to distinguish distributions from simple values, as a different execution machinery is needed for them. Another composite type, `DistributionTy`, summarizes the type of the elements a distribution generates. To keep the language simple we only allow distributions over simple types. We require that all vectors in `prpro` have constant fixed size, so we cannot use a distribution to specify a vector length. We see a vector as a special case of a distribution for which we also have observation data from experiments. When we manipulate a distribution and a vector we simply promote them to the results of manipulation of their element types.

To summarize, we construct a type language as a simplification (abstraction) of the typed language, considering what properties of the input elements we want to track. In our example, we track value ranges and lengths of vectors. The type language is implemented using the same mechanism as the abstract syntax of the typed language. In many simple languages, types do not have to be part of the language syntax. For statically typed languages, however, we need to allow users to include type annotations in models and programs. We typically also need to be able to print error messages, which may require pretty-printing type expressions.

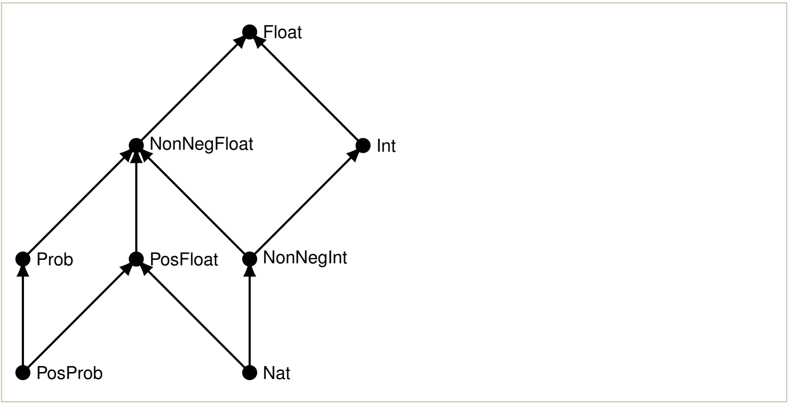


Figure 6.6: *Sub-typing hierarchy for simple types in* `prpro`*. There is an edge from type t_1 to type t_2 in the graph iff t_1 is a direct sub-type of t_2, written $t_1 \sqsubseteq t_2$*

This means that we also need a concrete syntax for types. One just includes them in the grammar definition of the typed language. Since defining the concrete syntax for types is not different from defining it for any other parts of the language, we skip the details in this chapter.

6.3 Type Hierarchy

Most type systems define a notion of *refinement* or *substitutability*. Substitutability means that when a model expects a value of a type t_1 at a certain syntactic location, it will also work correctly (or be meaningful) for any value of a sub-type t_2 of t_1 at this location [6]. For example, a probabilistic model written in `prpro` shall allow an integer number to be assigned where a floating point is required. More interestingly, we can use a probability distribution over values instead of a simple value in an arithmetic expression.

The simple types of `prpro` are organized in a hierarchy by inclusion between the sets of values they represent; a common, but not the only possible, criterion. See Fig. 6.6. Smaller types (like `Prob` representing probability) are below larger types (like `NonNegFloat` representing non-negative floating-point values). In the graph, we move downwards to sub-types and upwards to super-types. The lines going upwards connect types representing increasing sets of values. The largest simple type in our hierarchy is `Float`, and it includes `Int` as a sub-type, written `Int` $\sqsubseteq$ `Float`. In `prpro`, we will allow an integer at any position when a floating-point number is required. Similarly, positive probability is a more precise type than both probability (`Prob`) and positive floating-point numbers (`PosFloat`).

Formally, we interpret Fig. 6.6 as a partial order on simple types. The nodes positioned higher are bigger in the order, and the $\sqsubseteq$ symbol means "directly below in the graph." We generalize this for nodes that are not directly adjacent in the graph. We write $\sqsubseteq^*$ for the reflexive transitive closure of the relation $\sqsubseteq$. Thus $t_1 \sqsubseteq^* t_2$ means that t_1 and t_2 are connected by a directed path in the graph and the former lies below the latter; t_1 begins and t_2 ends a directed path. Types that are not connected by a directed path are *incomparable* and their values cannot be substituted, for instance `Prob` and `PosFloat`.

After defining a sub-typing hierarchy for simple types, we need to do the same for composite types: distributions and vectors. The main idea in `prpro` is that we can refine (substitute) values of simple types by distributions (to change usual calculations into calculations on random variables). Let us formalize these intuitions as sub-typing rules.

Each of the rules below has three parts: a name (in parentheses to the left), the premise (above the line), and the conclusion (below the line). The premise defines the condition that must be satisfied for the rule to be applicable. Multiple terms in a premise are interpreted conjunctively—they must all be satisfied. For instance, the premise of the very first rule, SSIMPLE, requires that two types, t_1 and t_2, are simple and that the former is a sub-type of the latter. Then the conclusion is that t_1 is a sub-type of t_2 also in our general sub-typing relation. We use the slanted inequality symbol ($\leqslant$) for the sub-typing ordering between arbitrary types, not just simple types. Do not confuse this symbol with the usual inequality symbol ($\leq$) representing the less-than-or-equal ordering on numbers. Here are all the sub-typing rules, with more commentary below:

$$(\textsc{SSimple})\frac{t_1, t_2 \text{ simple} \quad t_1 \sqsubseteq^* t_2}{t_1 \leqslant t_2}$$

$$(\textsc{SDist-1})\frac{t_1 \sqsubseteq^* t_2}{\mathsf{Distrib}(t_1) \leqslant \mathsf{Distrib}(t_2)} \qquad (\textsc{SDist-2})\frac{t_1 \sqsubseteq^* t_2}{\mathsf{Distrib}(t_1) \leqslant t_2}$$

$$(\textsc{SVect-1})\frac{l_1 \geq l_2 \qquad t_1 \sqsubseteq^* t_2}{\mathsf{Vector}(l_1, t_1) \leqslant \mathsf{Vector}(l_2, t_2)} \qquad (\textsc{SVect-2})\frac{t_1 \sqsubseteq^* t_2}{\mathsf{Vector}(l, t_1) \leqslant t_2}$$

The rule SVECT-1 relates vector types. We refine a vector type by sub-typing its element type *and* by ensuring that the refining type does not admit shorter vectors. This means that if a context in a model needs l_2 values of type t_2, then it will be able to operate on a prefix of a longer data set of l_1 elements, where each of the elements is also of type t_2 (because $t_1 \sqsubseteq^* t_2$). It might just ignore the excessive data elements. Note that the ordering on element types is consistent (in the same direction) as the ordering on vector types, while the ordering on lengths is inverted. The formal name for this phenomenon is type parameter *variance*. We say that element type here is a *co-variant* parameter of vector type (it refines in the same direction as the containing type), while length is *contra-variant* (it changes in the opposite direction).

A distribution type refines another distribution type if their element types are also sub-types (SDIST-1, co-variant). A distribution over elements of type t_1 cannot produce any values that a distribution of type t_2 would not be able to produce, at least in principle. The rules SVECT-2 and SDIST-2 rule handle the most controversial choice in the type system of `prpro`: it admits refinement of a simple type by a distribution type or a data set type. We want to allow a probability distribution or data in a location where a simple value is normally used in an expression. A distribution expresses uncertainty

```scala
def isSubTypeOf (t: Ty): Boolean = (this, t) match
  case (t1: SimpleTy, t2: SimpleTy) =>                // (SSimple)
    t1.superTys.contains (t2)
  case (VectorTy (l1, t1), VectorTy (l2, t2)) =>      // (SVect-1)
    l1 >= l2 && (t1 isSubTypeOf t2)
  case (VectorTy (l1, t1), t2: SimpleTy) =>           // (SVect-2)
    t1 isSubTypeOf t2
  case (DistribTy (t1), DistribTy (t2)) =>            // (SDist-1)
    t1 isSubTypeOf t2
  case (DistribTy (t1), t2: SimpleTy) =>              // (SDist-2)
    t1 isSubTypeOf t2
  case _ => false
```
source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/types.scala

Figure 6.7: A Scala implementation of the inductive definition of the sub-typing relation from p. 209

```java
public static Boolean isSubTypeOf (Ty t1, Ty t2)
{
  class SubTypeSwitch extends PrproTypesSwitch<Boolean> {
    public Boolean defaultCase (EObject t) { return false; }
  }
  return new SubTypeSwitch () {
    public Boolean caseSimpleTy (SimpleTy t1) {         // (SSimple)
      return new SubTypeSwitch () {
        public Boolean caseSimpleTy (SimpleTy t2)
        { return superTyTags (t1).contains (t2.getTag ()); }
      }.doSwitch (t2);
    }
    public Boolean caseVectorTy (VectorTy t1) {         // (SVect-1)
      return new SubTypeSwitch () {
        public Boolean caseVectorTy (VectorTy t2) {
          return t2.getLen () <= t1.getLen ()
            && isSubTypeOf (t1.getElemTy (), t2.getElemTy ());
        }
      }.doSwitch (t2);
    }
    ...
  }.doSwitch (t1);
}
```
source: prpro.java/src/main/java/dsldesign/prpro/java/Types.java

Figure 6.8: A fragment of the Java implementation of a sub-typing relation, rules SSIMPLE *and* SVECT-1 *shown on p. 209*

about this value. A vector represents experimental data about this value. What does it mean for the type system? We allow any simple type t_1 to be refined by a distribution generating values, or a data set containing values, of any of its sub-type. One can argue that distributions are a super-type of simple values, and we could organize this type system "upside-down." We feel that this results in a more complex type system, and does not follow the intuitions of Python's frameworks that inspired this example.

Exercise 6.1. Is `Float` the top type in the sub-typing hierarchy? In other words, is any other type in `prpro` a sub-type of `Float`? Analyze the sub-typing rules above to answer the question, and provide a proof, or a counterexample.

Exercise 6.2. Does there exist a single maximal type in the type hierarchy for `prpro` defined above? What is this type? If not, give examples of two types, and argue that they do not have a common super-type.

Implementation. Figure 6.7 shows the implementation of the above rules in Scala, following the same order as in the formalization above. In Line 3, we query a simple type for the set of its super-types to test the relation $\sqsubseteq$. Since the set of simple types is small and finite in `prpro`, we hard-coded the set of super-types (`superTys`) for each simple type. The remaining cases closely follow the formal definition of the rules.

The Java implementation is more complex. We show a fragment of it in Fig. 6.8. The structure and logic of the implementation is the same, but, since Java does not have pattern-matching expressions, we use a dynamic dispatch pattern with a `Switch` class generated by Ecore's infrastructure from the meta-model of the type language. This pattern allows computations to be split based on the abstract-syntax types. In lines 3–5, we define a switch instantiation for the task of sub-type checking. The idea is that a call to `SubTypeSwitch.doSwitch` produces a `Boolean` value: true if and only if `t1` is a sub-type of `t2`. The implementation of `doSwitch` is provided by Ecore. We need to define how to handle the individual cases of switching. We first override the method for the default case, stating that if no other rule has applied, then the sub-typing does not hold (Line 4). Then we instantiate the switch (lines 6–23), defining pattern matching on type `t1`. Since this pattern does not support matching on pairs of types, we use nested instantiations, which makes things harder to read. Still the traceability to the formal rules is fairly direct. The full implementation is available in prpro.java/src/main/java/dsldesign/prpro/java/Types.java.

6.4 Climbing the Type Hierarchy to Merge Compatible Types

We are typing an expression $e_1 + e_2$ where the sub-expressions are of types t_1 and t_2. What can we say about the type of the resulting value? A type checker decides which types are allowed to be combined. If the types are compatible, it computes their most precise common super-type. For instance, when adding an integer to a floating-point value, the result should be a floating-point number. The type describing the combination of types t_1, t_2 is known as the *join*, the *least upper bound*, or simply the *lub* of t_1 and t_2 in the sub-typing ordering. We write it as $t_1 \sqcup t_2$ in formal notation.

Without going into the mathematical details, the least upper bound of two simple types $t_1 \sqcup t_2$ is the type located above both t_1 and t_2 in the graph of Fig. 6.6, connected by directed path from both types, and the closest such (no shorter path can be found to a shared ancestor). We basically start climbing the hierarchy simultaneously from t_1 and t_2, and continue until the two paths meet. Figure 6.9 shows that `NonNegFloat` is an upper bound of `PosProb` and `Nat`, but the least upper bound is `PosFloat`:

$$\texttt{PosProb} \sqcup \texttt{Nat} = \texttt{PosFloat} \qquad (6.2)$$

A bigger question is: How shall we join composite types? For languages with simple type systems, like most DSLs, we can read this almost directly from the sub-typing rules. Since composite types are inductively defined,

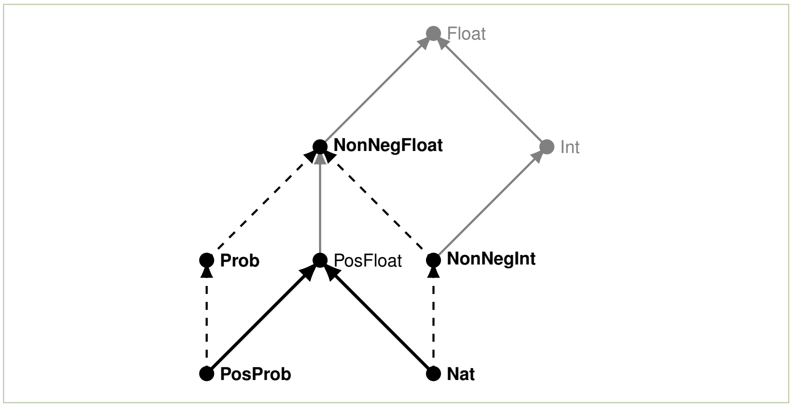


Figure 6.9: The type `NonNegFloat` *is an upper bound of types* `Nat` *and* `PosProb` *but* `PosFloat` *is the least upper bound*

this definition is going to be recursive. We are going to consider all possible pairings of types t_1 and t_2 and discuss how they should be combined, if at all. The definition is shown below in Eq. (6.3). We begin with simple types as a special case of composite types—we already know how to join them. We just delegate to Fig. 6.6.

$$t_1 \sqcup t_2 = \begin{cases} t_1 \sqcup t_2 \text{ in Fig. 6.6} & \text{if } t_1, t_2 \text{ are simple} \\ \mathsf{Vector}(\min(l', l''), t' \sqcup t'') & \text{if } t_1 = \mathsf{Vector}(l', t'),\ t_2 = \mathsf{Vector}(l'', t'') \\ \mathsf{Distrib}(t'_1 \sqcup t'_2) & \text{if } t_1 = \mathsf{Distrib}(t'_1),\ t_2 = \mathsf{Distrib}(t'_2) \\ & \text{if } t_1 = \mathsf{Distrib}(t'),\ t_2 = \mathsf{Vector}(l, t'') \\ & \text{if } t_1 = \mathsf{Vector}(l, t'),\ t_2 = \mathsf{Distrib}(t'') \\ t' \sqcup t'' & \text{if } t_1 = t' \text{ simple, } t_2 = \mathsf{Distrib}(t'') \text{ or } \mathsf{Vector}(l, t'') \\ & \text{if } t_2 = t'' \text{ simple, } t_1 = \mathsf{Distrib}(t') \text{ or } \mathsf{Vector}(l, t') \end{cases} \tag{6.3}$$

The second case specifies how to unify two vector types. The length of resulting vectors is the smaller of the lengths of the two joined types. The element type is a super-type (a lub) of the element types of the joined vectors. Compare these type transformations with the premises of rule SVECT-1. The longest vector that is no longer than both l' and l'' has $\min(l', l'')$ elements. The newly created vector type is a super-type for the combined types, but still as low in the sub-typing hierarchy as possible. This is consistent with the substitutability principle. We guarantee that any vector value correctly typed will offer at least as many elements as its type announces, perhaps more. Also, all the elements in the vector will be typeable with the inferred element type of the vector.

Similarly, when joining two distribution types (the third case), we join the element types and obtain a new distribution type that is the closest super-type as per rule SDIST-1. The final case deals with sub-typing along SDIST-2 and SVECT-2. The cases join a distribution type with a simple type, or with the element type of another complex type, resulting in the closest simple type above in the sub-typing hierarchy. Compare this with

```
 1 val topologicallySortedSimpleTys = List (NatTy, PosProbTy,
 2   NonNegIntTy, PosFloatTy, ProbTy, IntTy, NonNegFloatTy, FloatTy)

 4 def lub (t1: SimpleTy, t2: SimpleTy): SimpleTy =
 5   topologicallySortedSimpleTys
 6     .find { t => (t isSuperTypeOf t1) && (t isSuperTypeOf t2) }
 7     .getOrElse (null) // find always succeeds (null should never be used)

 9 def lub (t1: Ty, t2: Ty): Ty = (t1, t2) match
10   case (ty1: SimpleTy, ty2: SimpleTy) =>
11     lub (ty1, ty2)
12   case (VectorTy (len1, ty1), VectorTy (len2, ty2)) =>
13     VectorTy (len1 min len2, lub (ty1, ty2))
14   case (DistribTy (ty1), DistribTy (ty2)) =>
15     DistribTy (lub (ty1, ty2))
16   case (VectorTy (len1, ty1), ty2) =>
17     lub (ty1, ty2)
18   case (DistribTy (ty1), ty2) =>
19     lub (ty1, ty2)
20   case (ty1: SimpleTy, DistribTy (ty2)) =>
21     lub (ty1, ty2)
22   case (ty1: SimpleTy, VectorTy (len2, ty2)) =>
23     lub (ty1, ty2)
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/types.scala

Figure 6.10: *A Scala implementation of join (the least upper bound) for* `prpro`*: for simple types (lines 4–7) and for composite types (lines 9–23)*

rules SVECT-2 and SDIST-2, where a simple type is a super-type of a vector or a distribution if it is a super-type of its element type.

Implementation. For a small known set of simple types, like in `prpro`, the least upper bound can be pre-computed for any pair of simple types. However, pre-computing might be annoying in early design stages, when types are changing a lot. Every time you add or modify a type the pre-computed map needs to be updated. To avoid this problem, we sacrificed efficiency for flexibility, and proceed like with sub-typing of simple types: we sorted the types topologically and used a simple algorithm that walks up the sorting until we find the first node that is a super-type of both types combined. This way, we only needed to update the topological sorting in one place when updating the simple type hierarchy during development.

Figure 6.10 presents the implementation of type joining for both simple and composite types for the abstract syntax as a Scala ADT. In lines 1–2, we define a topologically sorted list of simple types. As expected, `FloatTy` is the very last type on the list, as it is also the top type in the hierarchy of simple types in Fig. 6.6. (Why is `NatTy` first?) Lines 4–7 show the implementation of lub for simple types: find the first type on the topologically sorted list that is a super-type of both `t1` and `t2`. This operation should never fail, so Line 7 never returns `null`. It is still needed to satisfy Scala's type checker, because the `find` function on lists returns an option of the identified value, not the value directly.

Finally, the `lub` function for general types (including the composite types) is shown from Line 9 onwards. It computes a join of two types. Lines 11–23 implement the cases in Eq. (6.3). In Line 11, we delegate to the `lub`

```
 1 protected static final List<SimpleTy> topologicallySortedSimpleTys =
 2   List.of (natTy, posProbTy, nonNegIntTy, posFloatTy, probTy, intTy,
 3     nonNegFloatTy, floatTy);

 5 public static SimpleTy lub (SimpleTy t1, SimpleTy t2)
 6 {
 7   return topologicallySortedSimpleTys
 8     .stream ()
 9     .filter (t -> isSuperTypeOf (t,t1) && isSuperTypeOf (t,t2))
10     .findFirst ()
11     .orElse (null); // never used
12 }

14 public static Ty lub (Ty t1, Ty t2)
15 {
16   class LubSwitch extends PrproTypesSwitch<Ty> {
17     private EObject ty;
18     public LubSwitch (EObject t) { this.ty = t; }
19     public Ty get () { return this.doSwitch (ty); }
20   }

22   return new LubSwitch (t1) {
23     ...
24     @Override public Ty caseVectorTy (VectorTy t1)
25     {
26       return new LubSwitch (t2) {
27         @Override public Ty caseVectorTy (VectorTy t2)
28         {
29           SimpleTy ty = lub (t1.getElemTy (), t2.getElemTy ());
30           int len = Math.min (t1.getLen (), t2.getLen ());
31           return vectorTy (len, ty);
32         }
33         @Override public Ty caseSimpleTy (SimpleTy t2)
34         { return lub (t1.getElemTy (), t2); }
35         @Override public Ty caseDistribTy (DistribTy t2)
36         { return lub (t1.getElemTy (), t2.getOutcomeTy ()); }
37       }.get ();
38     } ...
```

source: prpro.java/src/main/java/dsldesign/prpro/java/Types.java

Figure 6.11: *A Java implementation of join for simple and composite types in* `prpro`*; Only the case corresponding to* SVECT *is shown*

fuction for simple types. In lines 12–15, we apply the simple type join to element types, and the minimum function to the vector sizes. In lines 16–23, we promote one of the sides to a simple type, if none of the previous types matches, which effectively implements the last case from Eq. (6.3).

Figure 6.11 presents a small fragment of the corresponding Java implementation. Like in the Scala version, we first define the topological order of simple types (lines 1–3). Then we implement least upper bound for them, by searching the topological sorting. Finally, in lines 14–38, we show a fragment of the implementation of `lub` for composite types. Like in Fig. 6.8, we use the switch pattern classes generated by the Ecore infrastructure. First, we specialize the switch class for the `lub` computation (16–20). This computation is total, so we do not need to override the default case—another case would always match first. Second, we instantiate it and show how to merge two vector types, with the core operations in lines 27–32. The entire implementation is available from the book's code repository.

6.5 A Type-Checking Algorithm for Expressions

We have discussed relations between types (sub-typing) and operations on types (least upper bound). We are ready to talk about the actual type checking—relating types not to each other but to values and expressions in `prpro` models. We first assign basic numeric types to literals and constant values. To assign the most precise type for a constant we find the type placed lowest in the hierarchy of Fig. 6.6 that still contains the constant. A positive integer literal (say `42`) is assigned type `Nat`. Zero (`0`) is typed `NonNegInt`, as non-negative integers are the smallest of our types that include zero. All remaining integer literals (`-42`) are typed `Int`. Similarly, a positive floating-point literal (`3.14`) is typed as a positive float, unless it is a zero (`0.0`), which is typed as a probability. Any other number between zero and one is typed as a positive probability. Literals below zero (`-3.14`) are typed `Float`. This is summarized in the following definition:

$$\text{type-of}(c) = \begin{cases} \textsf{Nat} & \text{if } c \text{ is a positive integer literal} \\ \textsf{NonNegInt} & \text{if } c = 0 \\ \textsf{Int} & \text{for other integer literals} \\ \textsf{Prob} & \text{if } c = 0.0 \\ \textsf{PosProb} & \text{for } c \in (0;1] \\ \textsf{PosFloat} & \text{for floating-point literal } c > 1.0 \\ \textsf{Float} & \text{for other literals} \end{cases} \tag{6.4}$$

We assigned types to literals and constants directly, because their meaning is fixed and context-independent. In contrast, the type of an expression referring to variables depends on properties of these variables. A sum $x+y$ gives an integer if both x and y are integers. It is a float if both x and y are floating-point variables. Consequently, we need to know the types of smaller sub-expressions to type larger expressions. To capture this contextual information, we will store types of known variables in a *typing environment* denoted with the Greek letter Γ. An environment (Γ) is simply a map from variable names to types. It carries the information about types declared at various locations in the model to the places were the variables are used. Since `prpro` has data declarations as well as let-declarations, we need two environments. We will use Γ for storing the types of let-bindings, and will use Δ to store the types of the associated data sets. This way, we can have the same names both defined in the model (`let`, Γ) and supported by the data (`data`, Δ).

The typing rules assign a type (t) to each `prpro` expression (e) in typing environments (Γ and Δ). We will use the following ternary *typing judgement* to state this formally and concisely:

$$\Gamma, \Delta \vdash e : t \tag{6.5}$$

Like before, we will use this judgement in inference rules relating premises (above the line) and conclusions (below the line). We begin introducing the rules with the two simplest cases, the constant literals and variable references. The first rule below, CONST, defines the type for a constant c invoking Eq. (6.4). The second rule, VAR-REF-DATA, types a reference to a data set. The premise checks what type has been assigned to the variable `name` in the environment Δ and simply returns that type. The final rule, VAR-REF-LET, types a variable reference using a previous let-binding, provided that there is no corresponding data set (so data declarations have precedence in this type system). There is no other way to type a variable access in `prpro`. If a variable has not been typed (assigned) before accessing we will not be able to type it, which will result in a type error:

$$(\textsc{Const})\frac{c \text{ is a literal}}{\Gamma, \Delta \vdash c : \text{type-of}(c)} \qquad (\textsc{Var-Ref-Data})\frac{\Delta(\texttt{name}) = t}{\Gamma, \Delta \vdash \texttt{name} : t}$$

$$(\textsc{Var-Ref-Let})\frac{\Delta(\texttt{name}) \text{ is undefined} \qquad \Gamma(\mathit{name}) = t}{\Gamma, \Delta \vdash \texttt{name} : t}$$

The type of a binary arithmetic expression is inferred from the types of its sub-expressions. This means that a sum of floats will remain a float, and a sum of integers will remain an integer:

$$(\textsc{BExpr})\frac{\Gamma, \Delta \vdash e_1 : t_1 \qquad \Gamma, \Delta \vdash e_2 : t_2 \qquad t = t_1 \sqcup t_2}{\Gamma, \Delta \vdash e_1 \oplus e_2 : t}$$

The typing rule BEXPR above is unsound for some of our simple types. "Unsound" means that it can be used to conclude a type for a value that is inconsistent with the meaning of that type. Two examples:

1. Since `1` is of type `Nat` and `42` is `Nat` the typing rules allow us to conclude that '`(1 - 42): Nat`' (the result should be `Int`)
2. Since '`0.6: PosProb`' and '`0.42: PosProb`' we can conclude that '`(0.6 + 0.42): PosProb`' (it should be `PosFloat`; probability cannot exceed `1`).

There are several ways to make this rule sound. Perhaps the easiest is to assign a larger type than the least upper bound (respectively `Int` and `Float`) for results of the expression. This will make a workable type system, but we will loose all the fine granularity of numeric types that we so carefully designed. A more complex, but a more precise solution is to write a rule for each operator and numeric type separately. For instance, we do know that a sum of two natural numbers is a natural number, but for a difference we can only promise that it is an integer.

Exercise 6.3. Design a solution for this problem, and sketch the typing rules to return these types instead of $t_1 \sqcup t_2$ for the binary expression case.

Implementation. Figure 6.12 presents a Scala implementation of the rules BEXPR, CONST, VAR-REF-DATA, and VAR-REF-LET (Lines 5–23). The

```
 1 type Result[+T] = Either[String,T]

 3 def tyCheck(tenv: TypeEnv, denv: TypeEnv, expr: Expression): Result[Ty] =
 4   expr match
 5   case BExpr (left, operator, right) =>
 6     for
 7       t1 <- tyCheck (tenv, denv, left)
 8       t2 <- tyCheck (tenv, denv, right)
 9     yield expr.setTy (lub (t1, t2))

11   case CstI (n) if n > 0 => Right (expr.setTy (NatTy))
12   case CstI (0) => Right (expr.setTy (NonNegIntTy))
13   case CstI (_) => Right (expr.setTy (IntTy))
14   case CstF (0.0) => Right (expr.setTy (ProbTy))
15   case CstF (x) if x > 0.0 && x <= 1.0 => Right (expr.setTy (PosProbTy))
16   case CstF (x) if x > 1.0 => Right (expr.setTy (PosFloatTy))
17   case CstF (_) => Right (expr.setTy (FloatTy))

19   case VarRef (name) =>
20     denv.get (name)
21       .orElse (tenv.get (name))
22       .map (expr.setTy)
23       .toRight (s"Undeclared variable '${name}'")

25   case Normal (mu, sigma) =>
26     for
27       _ <- tyCheck (tenv, denv, mu).ensure (
28         t => t.isSubTypeOf (FloatTy),
29         t => s"Need a sub-type FloatTy for mu but got '$t'")
30       _ <- tyCheck (tenv, denv, sigma).ensure (
31         t => t.isSubTypeOf (NonNegFloatTy),
32         t => s"Need a sub-type of NonNegFloatTy for sigma but got '$t'")
33     yield expr.setTy (DistribTy (FloatTy)) ...
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/typeChecker.scala

Figure 6.12: *The type-checking rules for simple expressions (lines 5–23) of* `prpro` *and for a normal distribution node (lines 25–33) implemented in Scala*

signature of function `tyCheck` corresponds to the structure of the typing judgement in the rules: it relates typing environments (`tenv` is Γ, `denv` is Δ), an expression (`expr`), and a resulting type (`Result[Ty]`). The type `Result` (Line 1) represents a result of type checking: a `prpro` type inferred for an expression or an error message. The first case, implementing BEXPR, obtains the types of sub-expressions recursively. Then it combines them using the least upper bound. The use of the `for-yield` expression of Scala ensures that failures are propagated: if any of the type-check calls in lines 7–8 fails, then the entire `for-yield` fails and returns an error message.

Lines 11–17 implement the typing of literals based on the CONST rule and Eq. (6.4). These cases cannot fail—a type has been defined for any literal. We wrap the resulting type in the `Right` case of the `Result[Ty]`. (The `Left` case represents a failure.) Lines 19–23 implement the variable reference rules. First, the typing environment of data declarations (`denv`, Δ), a `Map[String, Ty]`, is queried for the type of the variable referred to by `name`. This results in a value of type `Option[Ty]`. If this does not succeed then the let-bindings environment is queried (`tenv`, Γ) following the design

```
 1 static class TypeEnvs {
 2   public Map<String, Ty> let;   // Gamma
 3   public Map<String, Ty> data;  // Delta
 4   ...
 5 }
 6 static class TyCheckExprSwitch extends TyCheckSwitch<Ty> {
 7   protected TypeEnvs env;
 8   ...
 9   @Override public Ty caseCstI (CstI expr)
10   {
11     Ty result = Types.intTy;
12     if (expr.getValue () > 0) result = Types.natTy;
13     else if (expr.getValue () == 0) result = Types.nonNegIntTy;
14     expr.setTy (result);
15     return result;
16   }
17   @Override public Ty caseBExpr (BExpr expr) throws TypeError
18   {
19     Ty t1 = tyCheck (this.env, expr.getLeft ());
20     Ty t2 = tyCheck (this.env, expr.getRight ());
21     expr.setTy (Types.lub (t1, t2));
22     return expr.getTy ();
23   }
24   @Override public Ty caseNormal (Normal expr) throws TypeError
25   {
26     Ty t1 = tyCheck (this.env, expr.getMu ());
27     if (!Types.isSubTypeOf (t1, Types.floatTy))
28       throw new TypeError (
29        "Need a subtype of FloatTy  for 'mu' but got '" + t1 +"'" );
30     Ty t2 = tyCheck (this.env, expr.getSigma ());
31     if (!Types.isSubTypeOf (t2, Types.nonNegFloatTy))
32       throw new TypeError (
33        "Need a subtype of NonNegFloatTy for 'sigma' but got '"
34        + t2 +"'" );
35     expr.setTy (Types.distribTy (Types.floatTy));
36     return expr.getTy ();
37   }
```

source: prpro.java/src/main/java/dsldesign/prpro/java/TypeChecker.java

Figure 6.13: *Selected type-checking rules for expressions of* `prpro` *implemented in Java, using the switch pattern with the infrastructure generated by Ecore*

of rules VAR-REF-DATA and VAR-REF-LET (l. 21). The `map` invocation stores the type in the annotation of the expression using a side effect for easy later access (l. 22). Function `setTy` is declared in Fig. 6.2. Finally, the option value is converted with `toRight` to a result of the `Right[Ty]` if this is successful. If it fails, `toRight` (slightly confusingly) creates a failing instance of `Left` encapsulating the provided error message. We encourage you to study the implementations of the normal distribution rule in lines 25-33. We will formalize the typing rule for these expressions below.

Figure 6.13 presents the key part of the corresponding Java implementation. A helper class `TypeEnvs` (l. 1–5) implements a pair of typing environments, for data and let bindings. These are stored in a local field during type checking (l. 7) and are accessed in the variable reference case, not shown in the figure. The functions (in l. 9, 17, and 24) process one meta-class from the meta-model, returning the inferred type. Instead of propagating errors using a special result type (as in our functional Scala example), we opt for using exceptions here, as a more natural Java idiom. Errors are propagated through the exception-handling control flow. Otherwise the design is similar to Fig. 6.12. We use the switch pattern classes of Ecore, providing an inner class to define visitors for various types, letting the Ecore generated machinery handle the dispatching based on the types of traversed syntax nodes. The implementation for BEXPR in lines 17–23 is almost identical to the Scala version. The implementation of CONST is split into two functions by the meta-model types; we only show one of them, for integers (l. 9–16). The entire implementation can be found in our source code repository.

We return to formal typing rules for `prpro`. The most domain-specific part of `prpro` expressions are the constructors of probability distributions. Typing them resembles typing other expressions, except for the distribution-specific type requirements for sub-expressions. The following rules formalize how to type them, both with and without an observation property. Recall that in `prpro` we can write a distribution expression just by invoking the name of the distribution, and providing the parameters. We can use constants and other variables as parameters, but also other distributions, and data sets.

$$(\textsc{Norm})\frac{\Gamma,\Delta \vdash e_\mu : \mathsf{Float} \qquad \Gamma,\Delta \vdash e_\sigma : \mathsf{NonNegFloat}}{\Gamma,\Delta \vdash \mathsf{Normal}(e_\mu, e_\sigma) : \mathsf{Distrib}(\mathsf{Float})}$$

$$(\textsc{Unif})\frac{\Gamma,\Delta \vdash e_0 : t_0 \qquad \Gamma,\Delta \vdash e_1 : t_1 \qquad t_0 \sqcup t_1 \leqslant t \quad t \text{ is simple}}{\Gamma,\Delta \vdash \mathsf{Uniform}(e_0, e_1) : \mathsf{Distrib}(t)}$$

The NORM rule states that a normal distribution expression always provides a distribution over floats. This is because a normal distribution assigns non-zero density to any real value. However, for typing to succeed, the type checker should prove that the mean parameter (e_μ) is a float (or a sub-type) and the standard deviation parameter (e_σ) is a non-negative float. The UNIF rule follows the same pattern, but for uniform distributions. The interesting aspect here is that we first type the endpoints of the interval (e_0, e_1), obtaining types t_0 and t_1. Note that these can be two different types. For instance, if the expressions are constant literals representing 0.42, and 42, then the first type is `PosProb` and the second one is `Nat`. To infer a single type of the elements generated by a uniform distribution over this interval, we can find the smallest simple type that includes the entire interval; in this case `PosFloat` (Fig. 6.9). We use the merging described in the previous section (lub) to find this type in the typing rules for the uniform distribution expressions.

Finally, we need to include automatic up-casting in the typing rules (which allows any type to be promoted to its super-type):

$$(\textsc{Upcast})\frac{\Gamma \vdash e : t_1 \quad t_1 \leqslant t_2}{\Gamma \vdash e : t_2}$$

The UPCAST rule allows a distribution to be used instead of a floating-point number for parameters of normal and uniform distributions. In the rules NORM and UNIF, the types t, NonNegFloat, and Float are simple. UPCAST exploits the sub-typing (DIST-2) to admit a distribution or a vector type in the same position. This way a distribution expression type checks even if it nests other distribution expressions and data set references.

Implementation. Our implementations of type checking of probabilistic expressions have been shown in Fig. 6.12 (lines 25–33) and in Fig. 6.13 (lines 24–37). In Scala, we use a method `ensure` implemented in `Result[T]` that takes a Boolean predicate on `T` and a function that formats an error message (typically both lambda expressions). The `ensure` function does nothing if applied to a result that is a failure (just propagates the left value). Otherwise it checks whether the predicate holds. If it does, `ensure` returns the value received, otherwise it formats an error message using the second argument and returns a failure (left) result with the same message. We show the type checking for normal distributions in both figures. We check the conditions one by one in the same order.

When we describe a type system formally, we tend to state properties declaratively: we specify what values and what types can be matched together. If you can prove, using the inference rules, that an expression types with type t, then you can prove the same for any super-type of t (substitutability). So the type system is *non-deterministic*.

Exercise 6.4. Study the formal typing rules and propose a small expression in `prpro` that can be typed both by $\mathsf{Vector}(5, \mathsf{PosProb})$ and by $\mathsf{Vector}(7, \mathsf{Prob})$. Prove both typings using our rules. Store your example for the next exercise.

In an implementation, the rules that update the typing environment have been made deterministic. When implementing a type checker we seek an *algorithmic* presentation, not a relational one. We achieve this by finding the smallest (the most concrete) type possible for every expression. We basically implement the most conservative interpretation of the typing rules and avoid using up-casting whenever it is not strictly needed.

Exercise 6.5. Study the implementation of one of our type checkers for `prpro`. What type will actually be returned for your example term from Exercise 6.4?

Finally, our implementation has no case corresponding to the UPCAST rule. This rule always introduces non-determinism. In the implementation, we basically replace type constraints (colon in the formal rules) with sub-type constraints ($\leqslant$) to allow relaxation in premises. This is still deterministic, because the sub-expressions are typed deterministically, and we just need to check whether their types are appropriate, directly or indirectly. See calls to `isSubTypeOf` for example in Line 29 of Fig. 6.12 and Line 27 of Fig. 6.13.

6.6 Type Checking for Models

Process the top-level declarations, and the typing of `prpro` will be complete. This task is much easier than type-checking expressions. Each `let` declara-

```scala
def tyCheck (tenv: TypeEnv, denv: DataEnv, decl: Declaration)
  : Result[(TypeEnv, DataEnv)] = decl match
  case Data (name, ty) =>
    val denv1 = denv.get (name) match
      case Some (_) =>
        Left (s"Data for '$name' has already been registered!")
      case None =>
        Right (denv + (name -> ty))
    denv1.map { denv1 => (tenv, denv1) }

  case Let (name, value) =>
    val tenv1 = tyCheck (tenv, denv, value)
      .ensure (
        t1 => tenv.get (name).isEmpty,
        t1 => s"'$name' has already been defined!" )
      .map { t1 => tenv + (name -> t1) }
    tenv1.map { tenv1 => (tenv1, denv) }
```
source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/typeChecker.scala

Figure 6.14: The type-checking rules for `prpro` *declarations, implemented in Scala. This function has to be put in a loop iterating over the entire model to complete the type checker*

tion sets the type of a name to the type of the right-hand-side expression; this type will be used for typing subsequent references to the variable. Multiple declarations are checked sequentially. Type checking of the entire model fails if any of them fails. In the formal rules below, DATA updates the data set environment Δ_0 with the type of a new data set for name. The rule ensures that name has not been previously defined, and captures the update in Δ_1:

$$(\textsc{Data})\,\frac{\Delta_0(\mathtt{name})\text{ is undefined}\qquad \Delta_1=\Delta_0[\mathtt{name}\mapsto \mathtt{t}]}{\Gamma,\Delta_0\vdash \mathtt{data\ name\ of\ type\ t}:\Gamma,\Delta_1}$$

$$(\textsc{Let})\,\frac{\Gamma_0,\Delta\vdash e:t\quad \Gamma_0(\mathtt{name})\text{ is undefined}\quad \Gamma_1=\Gamma_0[\mathtt{name}\mapsto t]}{\Gamma_0,\Delta\vdash \mathtt{let\ name}=e:\Gamma_1,\Delta}$$

$$(\textsc{Model})\,\frac{\Gamma_0,\Delta_0\vdash d_1:\Gamma_1,\Delta_1\quad\cdots\quad \Gamma_{n-1},\Delta_{n-1}\vdash d_n:\Gamma_n,\Delta_n}{\Gamma_0,\Delta_0\vdash d_1,\cdots,d_n:\Gamma_n,\Delta_n}$$

where d_i are declarations

The LET rule, processing variable-binding declarations, resembles DATA, but infers the type from the right-hand-side expression. Finally, type-checking the entire model requires that all declarations type-check correctly, accumulating the types and names in the two typing environments on the fly (MODEL). A `prpro` model is *well-typed* if we can use the above rules to type all the declarations according to the last rule.

Implementation. Like before, the implementation of the above rules should ensure determinism. Rule LET should not be satisfied with any type compatible with e, but should obtain the smallest, the most precise compatible type according to the sub-typing ordering. This makes it easier to reuse variables and expressions. If someone needs a Float and you give her a Prob, the model

```
1 @Override public TypeEnvs caseData (Data decl) throws TypeError
2 {
3   String name = decl.getName ();
4   if (this.env.data.containsKey (name))
5     throw new TypeError ("Identifier '" + name + "' already defined!");
6   env.data.put (name, decl.getTy ());
7   return env;
8 }

10 @Override public TypeEnvs caseLet (Let let) throws TypeError
11 {
12   String name = let.getName ();
13   Ty t1 = tyCheck (this.env, let.getValue ());
14   if (this.env.let.containsKey (name))
15     throw new TypeError ("Identifier '" + name + "' already defined!");
16   this.env.let.put (name, t1);
17   return env;
18 }
```

source: prpro.java/src/main/java/dsldesign/prpro/java/TypeChecker.java

Figure 6.15: The type-checking rules for declarations implemented in Java. This function has to be put in a loop iterating over the entire model to complete the type checker

will still make sense, but not if you give a Float when a Prob is expected. Fortunately, this is already guaranteed by the implementation of the type-checking rules for expressions. If the rules for expressions are made deterministic, then the model-level rules are deterministic, too. In the three formal rules above, only LET may introduce non-determinism, and only when typing an expression. No new non-determinism is introduced at this level. It is a good exercise to study the rules again to convince yourself about this.

Figures 6.14 and 6.15 show the implementations of LET and DATA. The former shows the entire implementation, while the latter only the overridden methods in the switch class for typing declarations. The DATA rule, in both examples, just checks for repeated declaration, and if no problem is found, records the declared type in the data set typing map. The LET rule first obtains the type of the right-hand-side expression. If successful, Line 14 (respectively 13) confirms that the variable has not been defined before. Either a failure of typing the expression or a repeated declaration of the same name cause the typing to stop with an error. Finally, the type environment is extended with a new definition (l. 16 in both figures) and returned (l. 17).

The MODEL rule uses a loop in Java and a fold in Scala (not shown for brevity). It processes the declarations one by one using the above-defined functions to build the typing environment. See our repository for details.

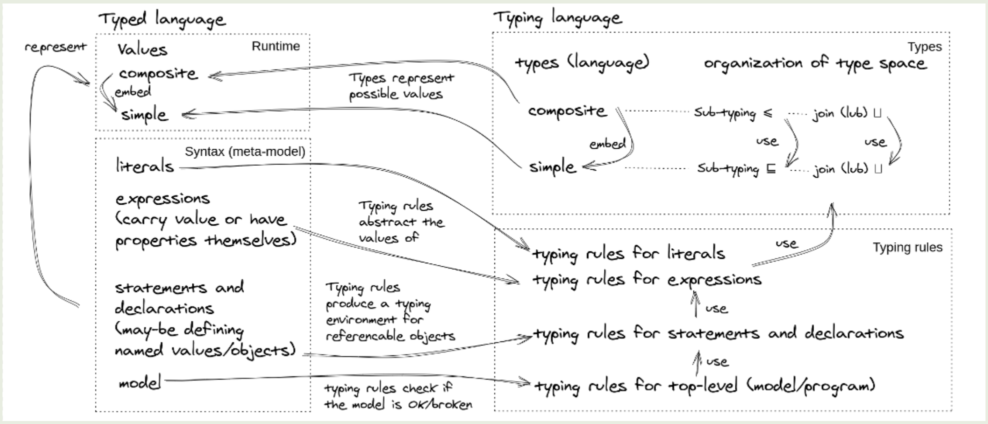
Exercise 6.6. Revisit the now complete implementation of the type checker. Which part needs to be modified to implement the solution to Exercise 6.3? Introduce the sound rule for binary expressions into the implementation.

6.7 Quality Assurance and Testing Type Checkers

Type systems are often developed using some formal specification (as we did), which gives the basis for developing systematic tests. To test a type system implementation, we test each component: each of the sub-typing rules, each of the join rules, and each of the type-checking rules.

What Do I Need to Build When Implementing a Type System?

The complexity of a type checker may overwhelm when compared to the terse constraints of Chapter 5. The diagram below summarizes the components of a typical implementation. Unlike a typical constraint, a type system examines the entire syntax instance, not just few related objects, to track non-local properties.


Typed language
represent
Values
Runtime
composite
embed
simple
Syntax (meta-model)
literals
expressions (carry value or have properties themselves)
statements and declarations (may-be defining named values/objects)
model
Types represent possible values
Typing rules abstract the values of
Typing rules produce a typing environment for referencable objects
typing rules check if the model is OK/broken
Typing language
Types
types (language)
organization of type space
composite
embed
simple
Sub-typing ⩽
Sub-typing ⊑
join (lub) ⊔
join (lub) ⊔
use
use
Typing rules
typing rules for literals
use
typing rules for expressions
use
typing rules for statements and declarations
use
typing rules for top-level (model/program)

The left column, *typed language*, lists syntax (Chapter 3) and runtime objects (Chapter 8) related by terms in the *typing language* (right column). Types replace values in an abstract interpretation, as if we computed on sets, not on concrete values.

typed language: `x + y`
typing lang.: Vector (200, Float)

Runtime is when a model is used computationally (not necessarily *run*). At runtime, concrete values arise: simple (numbers, strings, enumerations) and composite (objects, records, arrays, lists). Typically simple values are assigned simple types, and composite values are assigned composite types. The structure of the value domain is reflected in types.

simple type: Float
composite type: Distrib (Float)

We organize types into a *refinement hierarchy* resembling inheritance (sometimes exploited in implementations). If $t' \leqslant t$ then any value of type t should be *substitutable* by a value of t' without causing errors tracked by the type system. This is often done by making the set of values of t' be a subset of values of t.

distribution of non-negative floats is a float distrib.: Distrib (NonNegFloat) $\leqslant$ Distrib (Float)

When typing expressions, we often combine values of different types by up-casting them to a common super-type. This operation is captured by a join operation (least upper bound, LUB) on types, which has to be consistent with the sub-typing ordering.

NonNegInt $\sqcup$ Prob $=$ NonNegFloat and Vector(2,NonNegInt) $\sqcup$ Vector(4, Prob) $=$ Vector(2, NonNegFloat)

Types of simple literals are described by a direct case split. Inductive rules are needed if we have literals for composite values, and for expressions. A judgement decides what is the type of the value returned by an expression given the syntax of the expression and the context properties captured in a typing environment.

`1.0`: PosProb and `1`: Nat, `x+Normal(0,5)`: Distrib(Float) if $\Gamma(x) = $ Float

Statements, declarations, etc. update a typing context without carrying a type themselves, and may propagate multiple properties simultaneously. The typing environment stores information about referenceable properties that needs to be accessed later.

`let x=1` ensures that $\Gamma(x) =$ Nat if $\Gamma(x)$ undefined

At the top model-level we ensure the key Boolean property: Does the model type-check or not? We also store the entire type information collected during typing for use in the language implementation.

Scenario-driven testing. We create test cases for each rule, attempting to achieve good decision branch coverage. For each rule, find an input (abstract-syntax tree) satisfying the premises, and check whether the implementation types it as prescribed. The second column in Tbl. 6.1 shows examples of such test cases for `prpro`. For example, the first row has a test case derived from Fig. 6.6 to check whether the sub-typing implementation for simple types behaves as expected. The right column shows negative test cases, so examples of broken inputs to the type checker that violate the premises of the typing rule. For the sub-typing of simple types in the first row, we choose two types that are not sub-types in Fig. 6.6.

Often there are more ways to violate a typing rule than to satisfy it. Many negative test cases are needed to obtain good decision branch coverage. *Remember that a type checker is an error-finding tool, so testing whether it works is largely testing how it behaves on broken inputs.* In the SVECT-1 row of the table, we show a violation of the sub-typing rule for vectors. The

Table 6.1: Selected example test cases for the elements of the type checker; For each formal rule, test how it behaves when the input satisfies the premises, and when it violates them. Tests are shown in an informal dialect inspired by unit test matchers

rule	**positive test case**	**negative test case**

SSIMPLE	`PosProbTy.isSubTypeOf (PosFloatTy)`
	`  must be (true)`
		`PosProbTy.isSubTypeOf (NatTy)`
		`  must be (false)`

SVECT-1	`VectorTy (42, PosProbTy)`
	`  .isSubTypeOf (VectorTy(13,PosFloatTy))`
	`    must be (true)`
		`VectorTy (13, PosProbTy)`
		`  .isSubTypeOf (VectorTy (42, PosFloatTy))`
		`    must be (false)`

$t_1 \sqcup t_2$ for simple types	`lub (ProbTy, NonNegFloatTy)`
	`  must be (NonNegFloatTy)`
		no negative test (all pairs are unifiable)

$t_1 \sqcup t_2$ for composite types	`lub (VectorTy (10, intTy),`
	`     VectorTy (42, probTy))`
	`  must be (vectorTy (10, floatTy)`
		no negative test (all pairs are unifiable)

type-of(t) for simple types	`typeOf(CstF (0.6)) must be (PosProbTy)`	no negative test (all literals are assigned a type)

VAR-REF-LET	`VarRef ("x") with`
	`  tenv = Map ("x" -> PosProbTy),`
	`  denv = Map ()`
	`  is of type (PosProbTy)`
		`VarRef ("x") with`
		`  tenv = Map ("y" -> DistribTy(FloatTy))`
		`  denv = Map ("y" -> VectorTy(50,FloatTy))`
		`  fails to type-check`

NORM	`val e = Normal (CstF (0.42), CstF (0.1))`
	`tyCheck (tenv0,denv0,e) must be (FloatTy)`
		`val e = Normal (CstF (0.42), CstF (-0.42))`
		`tyCheck (tenv0,denv0,e) must fail`

LET	`Let("x", BExpr (CstI (1), Plus,`
	`  Normal(CstF(0.0),CstF(0.1))))`
	`  with tenv = denv = Map() must give`
	`  tenv = Map ("x" -> Distrib(FloatTy))`
		`Let ("x", BExpr (CstI (1), Plus,`
		`  Normal (CstF (0.0), CstF(0.1))))`
		`  with tenv = (Map ("x"-> PosProbTy)`
		`  must fail`

source: prpro.scala/src/test/scala/dsldesign/prpro/scala/adt/TypesSpec.scala
source: prpro.scala/src/test/scala/dsldesign/prpro/scala/adt/TypeCheckerSpec.scala
source: prpro.java/src/test/java/dsldesign/prpro/java/TypesTest.java
source: prpro.java/src/test/java/dsldesign/prpro/java/TypeCheckerTest.java

sub-type is shorter (13) than the super-type (42). Confirm with the SVect-1 rule that this is indeed a negative test case. Notice that this is not the only one way to violate SVect-1. For example, two types of the same lengths but incomparable element types would fail the sub-typing check, too.

The remaining rows in the table show examples for each rule category of this chapter: sub-typing for simple types, sub-typing for composite types, joining simple types, joining composite types, typing literals, and typing expressions (three rows). You can use this table to see whether you understand our typing rules or the implementation presented in the chapter. In an implementation, we incorporate them in automated unit tests and use them continuously in development. We add to them any regressions identified later in the project.

> **Exercise 6.7.** Design positive and negative test cases for some of the rules not shown in Tbl. 6.1: SDist-1, SDist-2, BExpr, SVect-2, Unif, and Data. Add them to tests for the Scala or Java implementation of the `prpro` type checker.

Testing the Model rule requires a larger input. It can be constructed from test cases for smaller parts, but it is more beneficial to obtain an independent test. Take the maximal example designed for testing the parser of your language, and evolve it into a type-correct example. (Often the maximal example for the parser is immediately a negative test case for the type checker.)

Property-driven testing. Scenario-based testing can get tedious when the different aspects of the language interact with each other, easily leading to a combinatorial explosion of the test case space. To test sub-typing for our eight simple types we need 64 test cases, one for each pair. This may appear overly conservative. For instance, we can easily cut the number of test cases in half, if we can establish general laws:

sub-typing is reflexive

```
forAll { (t: Ty) => t.isSubTypeOf (t) must be (true) }
```

sub-typing is anti-symmetric

```
forAll { (t1: Ty, t2: Ty) =>
  whenever (t1.isSubTypeOf (t2) && t2.isSubTypeOf (t1))
  { t1 must be (t2) }}
```

sub-typing is transitive

```
forAll { (t1: Ty, t2: Ty, t3: Ty) =>
  whenever (t1.isSubTypeOf (t2) && t2.isSubTypeOf (t3))
  { t1.isSubTypeOf (t3) must be (true) }}
```

join is a super-type of its arguments

```
forAll { (t1: Ty, t2: Ty) =>
   t1.isSubTypeOf (lub (t1,t2)) must be (true)
   t2.isSubTypeOf (lub (t1,t2)) must be (true)
}
```

join is the least super-type of its arguments

```
forAll { (t: Ty, t1: Ty, t2: Ty) =>
  whenever (t1.isSubTypeOf (t) && t2.isSubTypeOf (t)) {
      lub (t1, t2).isSubTypeOf (t) must be (true)
    }
  }
```

source: prpro.scala/src/test/scala/dsldesign/prpro/scala/adt/TypesSpec.scala

Table 6.2: Examples of property-based tests for the `prpro` type system. Note that these tests are the same for any type system. Code examples use the Scalatest library, but similar Quickcheck-style libraries exist for any mainstream programming language

1. For any simple type t we have $t \sqsubseteq t$
2. For any two simple types, if $t_1 \neq t_2$ we have that if $t_1 \leqslant t_2$ then $t_2 \nleqslant t_1$

This is what property-based testing is about. Instead of formulating discrete inputs, we formulate laws that should hold for large classes of inputs, and test these laws on many possible random values. Table 6.2 shows five essential property-driven tests for the `prpro` type checker. These tests have been derived from the fundamental properties of the type system that we want to be able to establish for any type system: namely that *sub-typing with join form a partial order*. The first row tests that a type is always a sub-type of itself, the second that two mutual sub-types must be equal. The third states that a sub-type of a sub-type is a sub-type as well (transitivity). The fourth checks whether a join of two types results in a super-type. The final row tests that the obtained super-type is the smallest possible.

The table uses the syntax of the Scalatest library, but many alternative libraries exist for all mainstream programming languages (e.g. Junit-quickcheck for Java,[3] hypothesis[4] for Python). Property-based testing is particularly useful for testing highly reusable components that are going to experience a diversity of inputs, like language tool chains. It helps to increase test coverage with automation.

The skill of writing property tests resembles writing constraints (Chapter 5). However, we constrain not the syntax of our language but its types and the behavior of the type checker. If you are used to writing static semantics constraints, you will easily succeed in writing property-based tests. Obviously, there are more properties than the generic five that one could write, also properties that are specific to the implemented language. We show more in the implementation of `prpro` in our code repository.

Property-based testing interacts well with scenario-based testing. Whenever a property-based test fails, the testing framework provides you with the failure-inducing input. Since the framework is randomized, it is prudent to store that input as a regression, besides using it for debugging. This way you will accumulate a collection of test cases quite fast.

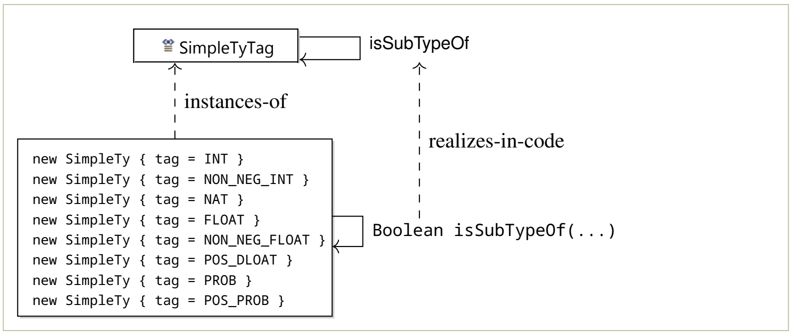
Property-based testing needs a way to generate inputs automatically. For all the types tested by properties, one needs to create generators compatible with the framework. Figure 6.16 shows the generator for our composite types. It is an inductive procedure that constructs larger terms from smaller terms, and uses randomization to decide which sub-types to instantiate. Similar generators can be implemented for the abstract-syntax trees. They will become useful again in testing later parts of the language infrastructure, for instance code generators (Chapter 9).

Exercise 6.8. Write a regression scenario test exposing the unsoundness of our typing rules (cf. Exercise 6.3). At this stage, we cannot argue for unsoundness of a typing rule yet. We can show that `0.6: PosProb` and `0.42: PosProb` leads

[3] https://github.com/pholser/junit-quickcheck
[4] https://github.com/HypothesisWorks/hypothesis

```
val genTy: Gen[Ty] = Gen.oneOf (genSimple, genComposite)
val genSimple: Gen[SimpleTy] = Gen.oneOf (topologicallySortedSimpleTys)
val genComposite: Gen[CompositeTy] = Gen.oneOf (genVector, genDistrib)
val genVector: Gen[VectorTy] = for
    len <- genInt
    elemTy <- genSimple
  yield VectorTy (Math.abs (len % 1000) + 1, elemTy)
val genDistrib: Gen[DistribTy] = genSimple.map {ty => DistribTy(ty)}
```

Figure 6.16: Simplified generators for the `prpro` *types to be used to test type system properties. We are using the generator framework of the Scalacheck library, compatible with Scalatest*



```
new SimpleTy { tag = INT }
new SimpleTy { tag = NON_NEG_INT }
new SimpleTy { tag = NAT }
new SimpleTy { tag = FLOAT }
new SimpleTy { tag = NON_NEG_FLOAT }
new SimpleTy { tag = POS_DLOAT }
new SimpleTy { tag = PROB }
new SimpleTy { tag = POS_PROB }
```

Figure 6.17: The subtyping hierarchy is typically not implemented using subtyping in the implementation language (a shallow embedding) but it is coded separately as a function checking whether values representing types are related (a deep embedding)

to `(0.6 + 0.42): PosProb`, but not that the latter is unsound. We would need to know how to execute our operators, which we have not implemented yet. However, we can write a syntactic regression test stating that the type of `0.6+0.42` should be a super-type of `PosFloat`. This test will fail until you solve Exercise 6.3.

6.8 Types in the Language-Conformance Hierarchy

The *typing* and the *typed* language are easily confused, even more so if the *typing* language is a *part of* the *typed* language. Also, the types of your language (`prpro`) and those of the implementation language (here Scala, Java, and Ecore) may appear perplexingly similar. It is essential to draw clear lines and understand how the involved languages and types relate to each other. In this chapter, we followed the *deep embedding* design: the types of `prpro` are *not* types of Java or Scala, but they are *values* (see Def. 10.7 in Sect. 10.1). Type-checking is just an algorithm operating on values in the implementation language. It relates values representing expressions and declarations to values representing types. A deep embedding is a common choice for implementation of types. A *shallow embedding* is the dual pattern, popular in code generators and in internal DSLs. We define and discuss it in Chapters 9 and 10.

In the implementation of `prpro`, each simple type is a distinct value of type `SimpleTy`, which is a Scala (Java) type representing all simple types. Figure 6.17 attempts to visualize this for the Java implementation. The simple types are tags in an enumeration, so they are simple values, instantiating the enumeration type `SimpleTyTag`. The class `SimpleTy` wraps the enumeration for minor technical reasons. The sub-typing hierarchy of Fig. 6.6 is not captured by Java sub-typing (inheritance) but it becomes an association between values. We have implemented this association not as

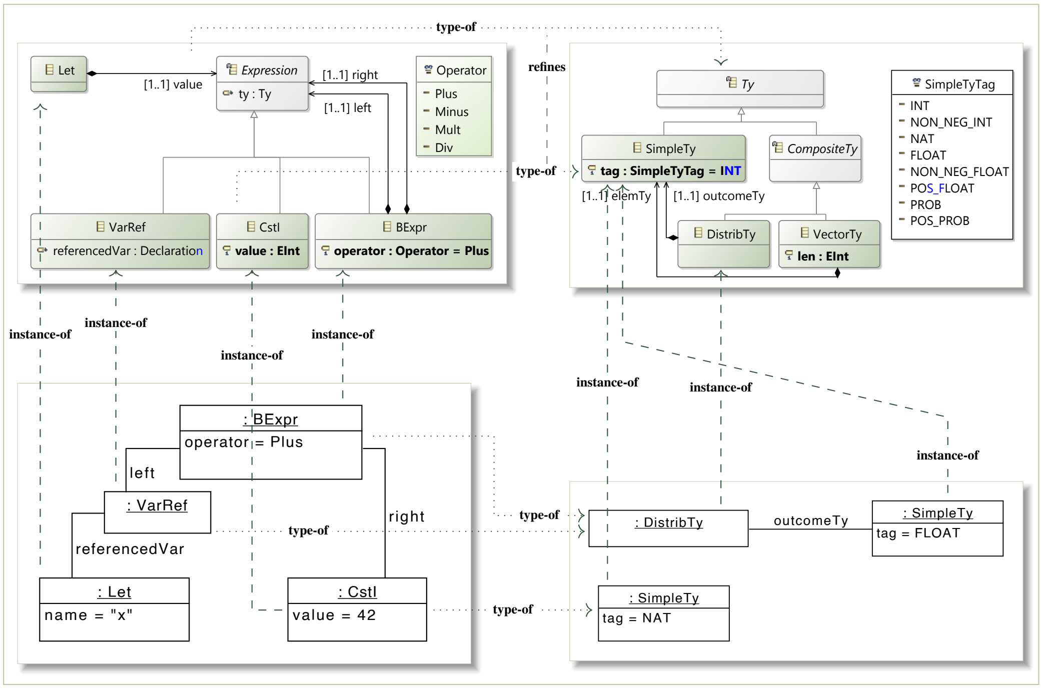


Figure 6.18: *Relating the typed language (top left), the typing language (top right), the typed language instance (bottom left), and the actual typing (bottom right). Dashed arrows represent instantiation (conformance), while dotted arrows represent typing*

a direct reference but as a function `isSubTypeOf` that derives the property from all super-types of each type (Fig. 6.8 line 10, and Fig. 6.7 line 3).

Figure 6.18 extends this overview to abstract syntax of models and composite types. We use Ecore for this example, which is easier to lay out visually than Scala. We are interested in typing a simple binary expression `x+42`, under the assumption that `x` is a distribution over floating-point numbers. We begin with the typed language, shown in the left part of the figure. The abstract syntax for the example is found in the bottom left corner. It follows the UML instance specification notation. The top left corner of the figure shows the relevant fragment of the `prpro` meta-model (quoting Fig. 6.3). The vertical dashed arrows connect each abstract-syntax object with the meta-class it instantiates. This left part of the figure is reminiscent of Fig. 3.13, except that only two-levels are shown, M1 and M2.

The right part of the figure presents the corresponding hierarchy for the *typing* language. The values in the bottom right corner represent types relevant for the example expression: the simple type of naturals (for the constant `42`) and the distribution over floats (for `x` and the resulting binary expression). These values are instances of meta-classes defining the syntax of the typing language shown in the top-right corner of Fig. 6.18, an exact copy of Fig. 6.5. Again, the vertical dashed lines mark the instantiation relations.

The dotted horizontal lines visualize the typing relations. At the instance level (bottom), concrete type values are assigned to each expression term

`VarRef`, `BExpr`, and `CstI`. These lines are reflected at the meta-level (top), which states that any expression object will be typed by a `Ty` object, adding further that constants shall be typed by `SimpleTy`. At the meta-level no concrete types are specified. We know that constants must have simple types, only because there is no way in `prpro` to write literal values for composite types. At the meta-level, concrete type assignments cannot be made. The type checker, operating on the instance level, can assign these types.

Mathematically, both the type-of relation and the instance-of relation are mappings *onto* simpler domains. The former maps to the domain of types, which is smaller and more abstract than the set of all models. The latter maps onto the meta-model, again a small set of meta-classes and relations. The former maps onto the types for the implemented language, the latter maps onto the types in the implementation language.

Further Reading

Programming language researchers have identified many use cases, engineering patterns, and sophisticated design methods for type systems that clearly go beyond the scope of this book. There are many formal problems to consider when designing a type system. Does there always exist a unique smallest type describing the value that can be produced by each expression? Or is the execution of a well-typed program going to preserve types (subject reduction)? What properties are guaranteed to hold for well-typed models (soundness)? How to design type systems that do not require explicit type annotations and support generic functions (parametric polymorphism) with type inference (solving type-variable constraints)? The classic introductory text on type systems that goes into considerable detail is the book of Pierce [10]. The goal of Lämmel [5] is closer to ours: to show the basics for application-oriented readers. Another book, with slightly more details about type systems, but still fairly efficient, is the programming language implementation book by Sestoft [11].

The goal of many researchers in type system engineering is to depart from manual implementation of type checkers, to create them automatically from high-level descriptions, in a way similar to how we generate parsers from grammars. Statix[5] [1, 9] is a tool that attempts to bridge type-checking and constraint solving. It provides a DSL for declarative specification of type correctness. Models in this DSL are automatically reduced to a constraint-solving problem. Statix is developed within the ecosystem of the Spoofax[6] language workbench [4]. Another tool for declarative definitions of type systems (and interpreters), integrated with Xtext, is Xsemantics[7] [2].

In `prpro`, we have developed an entire expression language for the purpose of the example. Many DSLs use a similar generic expression language. Xbase[8] is an implementation of a rich Java-like expression language with a type system provided by Xtext. It can be reused in other languages.

We do not teach property-based testing in this book but merely apply it to language implementations. To learn more about its pragmatics, search online for tutorials of ScalaCheck, QuickCheck (Haskell), Hypothesis (Python), or Junit-Quickcheck (Java). Property-based testing is an increasingly popular technique

[5] https://eelcovisser.org/research/#Statix
[6] https://www.spoofax.dev/
[7] https://github.com/eclipse/xsemantics
[8] https://www.eclipse.org/Xtext/documentation/305_xbase.html

that originated with the seminal paper of Claessen and Hughes [3] introducing the Haskell tool QuickCheck. Reading it is highly recommended. Since language definitions often have clear expected behaviors, it is natural to formulate laws that the implementations should adhere to—a natural playground for property-based testing [8]. Generating meaningful random models and programs remains a challenge, for instance ensuring that enough of them are well-typed [7].

Additional Exercises

Exercise 6.9. Add vector literals to the abstract syntax of `prpro` (the possibility of writing constant literals that represent vectors). Why does the typing hierarchy not need to be changed with this extension? Expand the typing rules to account for the new construct, and implement the new rule.

Exercise 6.10. Extend `prpro` with type-casting, so the ability to force a type of an expression. For instance, in Scala, we can add a constructor for expressions:

```
final case class Cast (e: Expression, ty: Ty) extends Expression
```

From the type-checker's perspective, a type-casting construct simply changes the type of the expression `e` to `ty` regardless of what the inferred type of `e` is. Adjust the type system to account for this new construct and implement the new rule.

This provides a workaround for the weaknesses of the numeric type inference in `prpro`. If the type system is unable to prove that a number is positive, the programmer may explicitly specify it. Of course, if the programmer is wrong, the model will be malformed. It would be prudent to insert a runtime check in the interpreter for `prpro`, to ensure that the type cast is safe for the value produced by *e*.

Exercise 6.11. Our type checker accepts inconsistent types of data and variables, but our intention is that the data set for a variable has the same type as elements in a distribution type of the same variable. Change the rule LET (p. 221), so that it behaves like now if there is no data definition for the variable, but if there is a data set for the variable, the type of elements in the data set has to be a sub-type of the elements in the distribution type matched. The right-hand side in a let expression must then be a distribution. See the next exercise for the implementation.

Exercise 6.12. Reimplement the type-checking function for declarations, either in Scala or Java, so that for LET it behaves like now when there is no data definition for the variable. If there is a data set for the variable, the type of elements in the data set has to be a sub-type of the elements in the distribution type matched.

Exercise 6.13. Implement two constraints enforcing the following properties: **a)** An expression can only refer to data sets (yielding a vector type locally) in the very last let binding in the model. The very last entry in the model should be a let binding. **b)** If a data set is referred to in a let binding's expressions, then there has to be a data set defined for the left-hand side variable in the binding, and both data sets (vectors) should be of the same size. Note that many interesting constraints become easy to write when we have inferred types for model elements.

Exercise 6.14. Add a Bernoulli distribution expression to the `prpro` meta-model and the type checker. A Bernoulli distribution has one parameter *p*, which takes value between zero and one (a probability value), and produces a distribution of Booleans. For our purposes, make it produce distributions of non-negative integers (to cover zero and one). Add a suitable type-checking rule for Bernoulli expressions summarizing the above specification, and implement it in Scala or Java. Take inspiration from normal and uniform distribution expressions.

Exercise 6.15. Add a Boolean type to `prpro` and its type system. The most natural way in a statistical language is to consider Booleans to be integers 0 and 1, as then we can talk about an average value of a series of Booleans. Thus make Boolean a sub-type of non-negative integers. Revise the typing rules in Eq. (6.4) accordingly. If you solved Exercise 6.14 then you can now revise the solution to make the new Bernoulli expression return a distribution of Booleans.

Exercise 6.16. Add two-dimensional vector types to the type language of `prpro` (so two-dimensional arrays, or vectors of vectors) and the typing rule to support it.

Exercise 6.17. (A small project) Add arbitrarily nested vector types to `prpro`. We can assume that the multidimensional vector values "enter" the language via the `Data` construct (we can still ignore how they are actually specified in external files). This extension requires revising the language of types, adding the sub-typing rules, join rules, and the typing rules.

Exercise 6.18. (A small project) Change `prpro`'s `Data` bindings for data sets to include a URL of a CSV file containing the data, instead of a type (Figures 6.2 and 6.3). Implement a simple inference tool that detects the type of the entries in the CSV file by their syntax and calculates the vector type returned by the type checker for data bindings based on this information. Integrate this inference into the type checker (Fig. 6.14) for `prpro` and test that it works well.

Exercise 6.19. (A small project) Extend the syntax of `prpro` with normal distributions that take a vector of numbers for the mean parameter `mu`. Such a normal expression should produce a vector of normal distributions, one for each value of the mean. (A "vectorized" distribution construct is common in Python libraries for probabilistic programming.) Extend the type language to allow vectors of distributions. Make sure that there is a new typing rule for normal distributions, and revisit all sub-typing, join, and typing rules for vectors, amending as needed.

Exercise 6.20. (A project) Reimplement the type system for `prpro` using XSemantics or Spoofax/Statix, and reflect on the added value of using a DSL-driven (model-driven) tool for type system implementation. When is it beneficial?

Exercise 6.21. (A small project) Revisit the finite-state-machine language from Chapter 3. Add global numeric variables, expressions (both arithmetic and Boolean), and assignments to the abstract syntax. Allow adding Boolean expressions as guards on transitions and assignments as actions. A transition is active and can be taken if the source state is active and the guard condition evaluates to true. Design a type system that ensures that only Boolean expressions, not numeric expressions, are used as guards on transitions.

Exercise 6.22. (A project) Design a simple language for electric circuits. Each wire in a circuit can carry DC or AC power. We have power inputs, and power outputs. Wires have two ends that can be connected to other wires or junctions. Junctions connect several incoming wires to outgoing wires. Finally, a frequency converter (one input, one output) changes incoming AC current into a DC current. Design and implement a type system which, given a typing environment that assigns AC or DC current to each input node for the circuit, infers the AC/DC current type for every wire and output node. A circuit should fail to type-check if AC and DC wires are connected without a frequency converter node.

References

[1] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A constraint language for static semantic analysis based on scope graphs". In: *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, 2016 (cit. p. 229).

[2] Lorenzo Bettini. "Implementing Java-like languages in Xtext with Xsemantics". In: *Symposium on Applied Computing*. ACM. 2013 (cit. p. 229).

[3] Koen Claessen and John Hughes. "QuickCheck: A lightweight tool for random testing of Haskell programs". In: *International Conference on Functional Programming*. ICFP. 2000 (cit. p. 230).

[4] Lennart C.L. Kats and Eelco Visser. "The Spoofax language workbench". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. SPLASH/OOPSLA 2010. ACM, 2010 (cit. p. 229).

[5] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018 (cit. p. 229).

[6] Barbara H. Liskov and Jeannette M. Wing. "A behavioral notion of subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841 (cit. p. 208).

[7] Jan Midtgaard and Anders Møller. "QuickChecking static analysis properties". In: *Softw. Test. Verification Reliab.* 27.6 (2017) (cit. p. 230).

[8] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. "Testing an optimising compiler by generating random lambda terms". In: *International Workshop on Automation of Software Test, AST 2011*. Ed. by Antonia Bertolino, Howard Foster, and J. Jenny Li. ACM, 2011 (cit. p. 230).

[9] Daniël A.A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. "Towards language-parametric semantic editor services based on declarative type system specifications". In: *European Conference on Object-Oriented Programming, ECOOP*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. 2019 (cit. p. 229).

[10] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002 (cit. pp. 202, 229).

[11] Peter Sestoft. *Programming Language Concepts*. Springer Science & Business Media, 2012 (cit. p. 229).

The only thing constant in life is change.
François de La Rochefoucauld

7 Model and Program Transformation

So far, we focused on defining the syntax of DSLs in efficient ways. We worked with abstract and concrete syntax. We have seen tools that can transform syntax definitions (meta-models and grammars in our case) not only into model editors, but into a whole infrastructure for processing models that adhere to the syntax definition.

The language infrastructure allows users to instantiate languages and create valid models. It serializes and de-serializes (i.e., parses) models, so that users can build and process the models through editor tools that conveniently support editing models in their concrete syntax. This infrastructure allows creation of models in dedicated languages and to their manual use for a variety of purposes, such as documentation, brainstorming, or the sharing of knowledge among users (e.g., developers, modelers, domain experts, or business experts)—essentially since we can store valuable data as models. We can even document the meaning of models textually. However, just manually using and processing models would limit their power substantially. In disciplines such as data management, which focus on efficiently storing data in various forms, this might be sufficient.

Recall that one of our main goals is *automation* of software engineering tasks. If we want to automate based on models, we will need to explicitly express the meaning of models—that is, their semantics—in a way that automated tooling can use. We already presented ways to explicitly express the static semantics. In this chapter, we will turn our attention more towards the dynamic semantics of languages.[1] Semantics give meaning to models and to their languages (i.e., meta-models). We usually define semantics for languages, which then also gives semantics to all their models.

The standard way to give dynamic semantics to programming languages (GPLs) or modeling languages (DSLs) is to write an interpreter or a compiler. In this chapter, we focus mostly on compilers, which translate models within or across languages. We discuss interpreters later in Chapter 8.

Transforming software artifacts is a very common task. Consider a traditional compiler, which processes source code. After recognizing the syntax, it performs many different transformations to translate the source code into an instance of another language, such as assembler or machine language, that can be executed. The instances are merged, split, optimized, and translated, which the compiler can do because it implements the language's semantics. In other words, specifying how models can be

[1] Recall our introduction to dynamic semantics from Sect. 2.4.

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_7

translated, transformed, processed, merged, split, and so on gives semantics. Models often have multiple semantics, but some are more important for the intended uses of the models than others.

Transforming programs or models is called *program transformation* or *model transformation*. Both are a form of *meta-programming*, so programming that treats other programs (or models) as data. In that sense, it is a more advanced programming activity than usual programming. Program and model transformation are very close, as models and programs are sometimes hard to distinguish.[2] The former is older and originates from the programming language community, which focused on building compilers. There, program transformations aim at optimizing programs, transforming (intermediate) program representations into other representations, essentially translating within or across languages. Model transformation originates from the software engineering field and mainly originates from the need to express software in domain-specific languages. The idea is to automate software engineering tasks and allow developers to work on higher levels of abstraction.

Countless publications about model and program transformation exist [39, 21, 70]. They contributed many important concepts realized in dedicated transformation languages and tools. As we will illustrate, transformations can be realized in contemporary programming technology as well. In fact, our experience shows that in practice, transformations are often realized using ordinary GPLs. Specialized transformation languages often require experts, who are difficult to hire, and when they leave a company, the maintenance of these transformations becomes challenging.

In this chapter, we focus on examples, definitions, classifications, practical guidelines, and quality assurance of transformations. We describe transformations written in the (arguably) mainstream programming language Scala and briefly illustrate some other, specialized transformation languages. All our transformations could be written in Java as well, but they would be more verbose. We will even show a transformation written in C that we found in the Linux kernel (Fig. 7.17), illustrating the verbosity of transformations when a language does not have convenient transformation abstractions. On a final note, we use the terms *model* and *program* synonymously. We could have just used model, but to avoid confusion we still use program when we talk about program-transformation techniques.

7.1 Technological Spaces

In the book, we largely tried to provide a problem-oriented and conceptual view of engineering DSLs. But, of course, in practice you will need to choose an actual implementation technology. We did meta-modeling using Eclipse's modeling framework EMF, realized concrete syntax in the EMF-compatible Xtext framework, and otherwise tried to realize everything else

[2] Recall our box on models versus programs on p. 29 in Chapter 2.

(e.g., static semantics, dynamic semantics, internal DSLs) in an arguably mainstream programming language—Scala. Over recent decades, many different technological spaces [41, 24] arose, many of which offer tools to realize all parts of a DSL. Sticking to one such technological space has many benefits, especially that the tools interoperate quite well, but it can also be limiting. You always rely on a vibrant community that keeps the tools updated, and there can easily be a mismatch between the technological space of the DSL and the rest of your system, requiring intricate integrations with glue code and bridges. In this context, our book tries to remain mainstream as much as possible with Scala, but leverages the powerful technological spaces for DSL engineering, especially Eclipse's EMF.

Inspired by Mens and Van Gorp [48], as well as Kurtev, Bézivin, and Aksit [41], we define a technological space as follows.

Definition 7.1. *A technological space is a set of well-integrated concepts, tools, mechanisms, and languages building upon a common technological platform. The platform is determined by a particular meta-modeling language.*

The space can be seen as a working context. It usually has an associated body of knowledge and a community that shares know-how and contributes to the space. The meta-modeling language in fact influences the space substantially. For instance, Ecore is based on class modeling, so you will see classes and objects together with their different relationships in many aspects in the space, determining the way of thinking and working within a space called *Modelware*.

In this book, so far, we mainly covered the technological spaces called *Modelware* and *Grammarware*. The main reason is that they aim at developing software, as opposed to spaces such as *SQLware*, which aims at managing large data, or *XMLware*, which aims at exchanging data.

Modelware focuses on object-oriented modeling techniques and arose from systems and software engineering. The observation was that one can use object-oriented modeling, specifically class diagrams, to describe the structure of other models—the terms model, meta-model, and meta-meta-model were born. Class diagrams are expressive, have a widely known concrete syntax (we use it throughout the book), and provide an intuitive modeling paradigm, sometimes called the lingua franca of software engineering. Originally conceived as a simple language to exchange ideas and brainstorm, that they became a precise modeling notation that is the basis of automated tools is actually contested,[3] but we believe that this use is where class diagrams shine when used right, especially when not used to describe whole systems, but focusing on domain-oriented aspects. Modelware also became popular due to heavy standardization efforts by the Object Management Group (OMG) with the MOF (Meta-Object Facility) and

[3] https://twitter.com/grady_booch/status/1388930413280727042

UML (Unified Modeling Language) standards. Many model-transformation technologies exist for Modelware; see Sect. 7.6.

Grammarware focuses on grammar-oriented descriptions of structure and arose from functional programming. The community interprets grammar broadly, as a "structural description in software systems" [40], so not necessarily a grammar in the sense of a parser specification—the latter is seen as an enriched grammar by proponents of Grammarware. A grammar should allow creation of instances of different types, should have constructs to compose the types and the instances into more complex structures, and should have constructs to describe choices among types and instances. Algebraic data types belong in this space, as well as context-free grammars, tree grammars, and graph grammars. Proponents of Grammarware call for more engineering principles, since grammars—descriptions of structures—exist in pretty much any software system. Compared to Modelware, a core difference is that models can describe non-structural aspects (e.g., behavior), while grammars are restricted to structure, and meta-models focus on describing language constructs for modeling, so meta-models contain grammars, while some grammars are meta-models. This illustrates that both meta-models and grammars are well suited to describe DSLs, as we do in this book. Many program-transformation techniques exist for Grammarware, specifically for algebraic data types; see Sect. 7.6.

Other technological spaces exist in addition to Modelware and Grammarware. Lämmel [43] lists the following ones together with their core technologies: *XMLware* (e.g., XML, XML infoset, DOM, DTD, XML Schema, XPath, XQuery, XSLT), *JSONware* (e.g., JSON, JSON Schema, JSONata), *SQLware* (e.g., table, SQL, relational model, relational algebra, WOL), *RDFware* (e.g., resource, triple, Linked Data, RDF, RDFS, OWL, SPARQL, STTL), *Objectware* (e.g., objects, object graphs, object models, state, behavior, visitor pattern), and *Javaware* (e.g., Java, Java bytecode, JVM, Eclipse, JUnit). They all come with their own transformation techniques, such as XSLT, JSONATA, WOL [22], STTL [18], while in plain object-oriented development one can use the visitor pattern and recursive functions to traverse models.

Model transformations operate within or across technological spaces. As you will see in our examples, the transformation language can easily be in a different technological space than the models that you want to transform. We use Scala to transform models expressed in Ecore, which, despite Scala being close to Java, requires some conversions (mainly of Java collections to Scala collections). We show that it is not too difficult to bridge these technological spaces, so that you can stick to arguably mainstream technologies.

7.2 Model-Transformation Case Studies

Let us jump directly into examples to illustrate very common use cases of model transformations: transforming models along different languages and

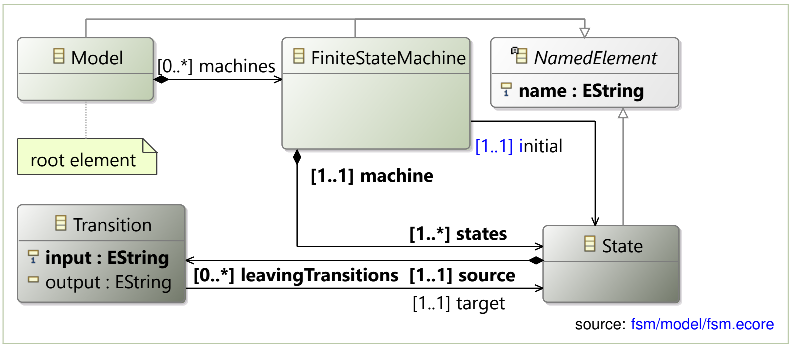


Figure 7.1: The meta-model of the `fsm` language, repeated from Fig. 3.1

along different versions of one language. In a later chapter, specifically in Sect. 13.1, we will see another transformation that configures variable Ecore models based on a configuration.

Case Study 1: From Finite-State Machines to Petri Nets

We first transform models from one format (i.e., language) into another format (i.e., language). The following is a small, but sufficiently involved example of translating between meta-models, which we use to demonstrate the various aspects of transformations.

Recall the `fsm` language, specifically its meta-model in Sect. 3.3, which expresses finite-state-machine models. We repeat it here in Fig. 7.1 and also show an example model in visual syntax for illustration in Fig. 7.2. We created this syntax using Sirius (cf. Sect. 2.2).[4] Our example model is a finite-state machine of a coffee machine that is operated through coins and allows the user to select either coffee or tea, and then goes into the state of brewing coffee or brewing tea. The other transitions among the coffee machine's states should be self-explanatory.

Transformation goal. Our goal is to transform `fsm` models into so-called Petri nets. The example is inspired from a model transformation by Hinkel [35]. As Hinkel explains, such a transformation is not too realistic, since in a real system, either one of the two formalisms is typically used, but not both. We choose this transformation as our demonstration example, since the two formalisms are well understood and conceptually not too far apart.

Petri nets [58] is a formal modeling language typically used to describe processes with parallelism. Without digging into the details and the many variants and extensions that exist for Petri nets, let us briefly define what a Petri net is. A Petri net is a directed bipartite graph that contains two kinds of nodes: places and transitions. Places contain tokens and are connected with each other via transitions. Petri nets prescribe the way tokens can be "fired" through transitions from one place to another, thereby simulating the dynamics of the system that the Petri net models. The exact semantics are not important for us, and we refer to the literature for details [58].

[4]The Sirius visual editor definition is available at fsm.sirius.design/description/sirius.odesign in our online repository.

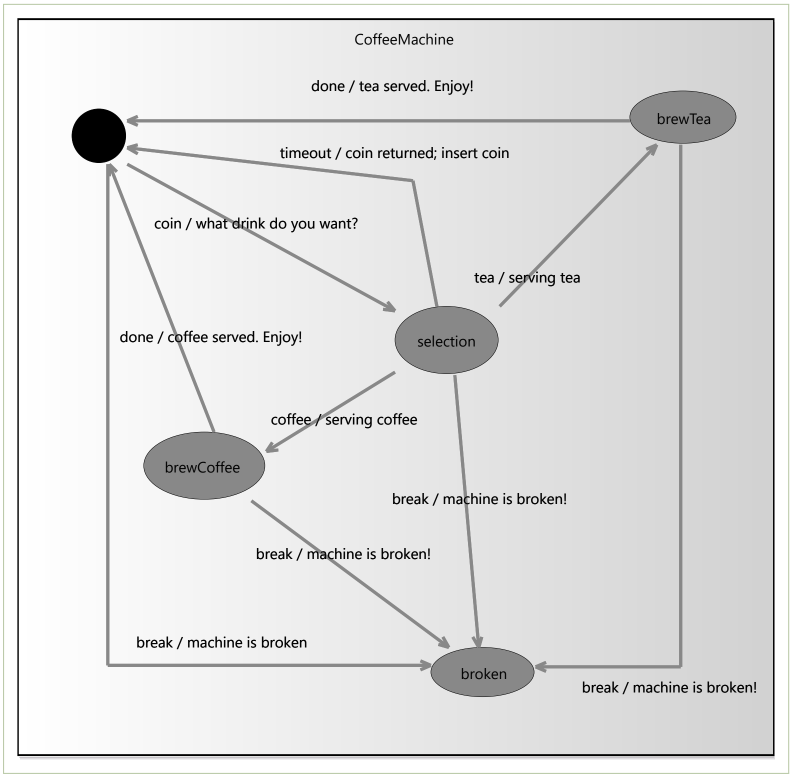

CoffeeMachine
done / tea served. Enjoy!
brewTea
timeout / coin returned; insert coin
coin / what drink do you want?
tea / serving tea
selection
done / coffee served. Enjoy!
coffee / serving coffee
brewCoffee
break / machine is broken!
break / machine is broken!
break / machine is broken
broken
break / machine is broken!

Figure 7.2: *A finite-state machine representing a simple coffee machine. See also Fig. 9.17 on p. 353*

A meta-model for our Petri nets language (referred to as `petrinet` in the remainder) is shown in Fig. 7.3, and an example in concrete syntax can be found in Fig. 7.4. The latter model is actually the result of transforming the coffee machine model from Fig. 7.2 into `petrinet`.

Transformation requirements. Our main requirement when transforming `fsm` models into `petrinet` models is that source and target model share the same execution semantics. In other words, the possible state transitions of a model should be equal to the possible transitions of tokens through places. The mapping is relatively simple. For each state in the state machine, there should be a place with the same name in the Petri net. For each transition in the state machine, there should be a transition in the Petri net, connecting exactly the places whose origin-states (identified by names) are connected in the state machine. There should be an initial token in the place with the same name as the initial state, and for each end state (i.e., a state that has no outgoing transition), there should be an additional transition that throws away a token (realized using a dangling transition, as shown for place “broken” in Fig. 7.4).

Transformation technology. We implement this transformation in four languages: Scala, Xtend, QVT-O, and ATL. We discuss the Scala transformation here, and beyond showing small snippets in Tbl. 7.1 for the other

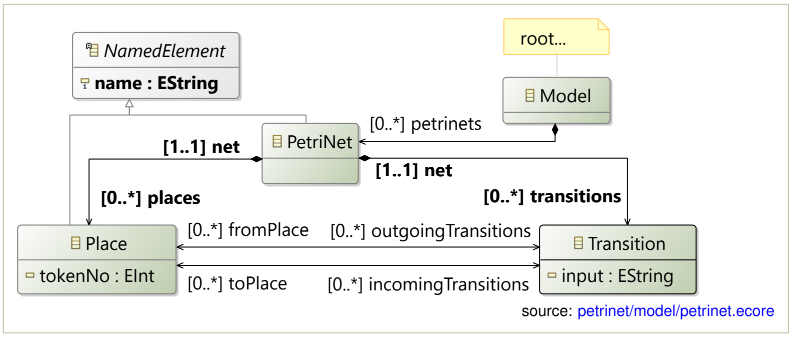


Figure 7.3: The meta-model of the simple `petrinet` *language*

languages, we refer to our online repository: fsm.xtend, fsm.qvto, and fsm.atl, as well as for the full implementation of the Scala transformation fsm.scala (file FSMToPetriNet.scala). The repository also contains the driver code that initializes the relevant Ecore library as well as loading the source model and saving the transformed target model.

Using different technological spaces—Eclipse EMF to represent source and target models and Scala for the transformation—has some implications.

First, you need to understand that, while we transform models, the transformation logic is implemented against the languages, more precisely, the meta-models. So, it is helpful to understand the transformation on examples, which you can then use to experiment with your implementation and eventually also validate it. As such, the implementation refers to meta-classes defined in the meta-models. In our case, these meta-classes are generated from Ecore models. Since the generator is not available for Scala, it generates Java classes, so we cannot exploit all the neat features of Scala, such as pattern matching, for transformations when using the Eclipse modeling framework (EMF) for handling source and target models.

Second, we need to convert Java collections into Scala collections and vice versa (see the many `.asScala` or `.asJava`. We could to some extent use Scala implicits, which would need to be declared for various meta-classes separately and, in our experience, make the transformation code even harder to maintain and evolve. We use Scala implicits to specify helper functions (also known as queries), however. Another alternative would of course be to write the whole transformation in Java, which also offers `map` and `filter` with closures on lists, which we commonly use in our Scala transformation.

Instead of representing the source and target languages and objects in Ecore and then accessing the generated Java classes from Scala, we could have represented the languages using ADTs (algebraic data types) in Scala, more precisely, Scala case classes. This would have avoided all the .asScala and .asJava calls, but as already discussed above, would deprive us of all the other benefits of using EMF, such as free persistence and many other frameworks, such as Xtext. Alternatively, we could have written the transformation in Java or in one of the dedicated transformation languages available for the Eclipse EMF technological space (e.g., Xtend,

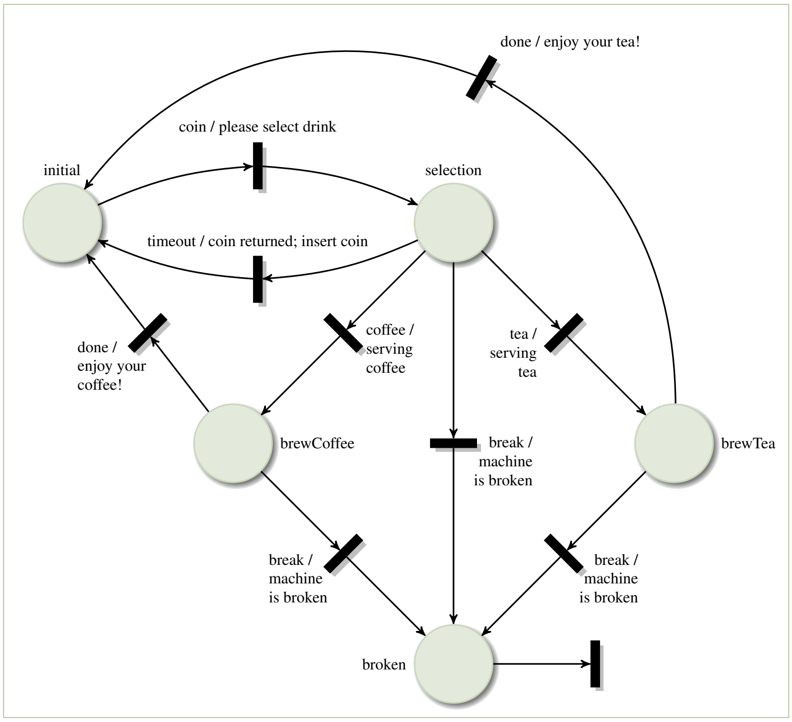

done / enjoy your tea!
coin / please select drink
initial
selection
timeout / coin returned; insert coin
done / enjoy your coffee!
coffee / serving coffee
tea / serving tea
brewCoffee
break / machine is broken
brewTea
break / machine is broken
break / machine is broken
broken

Figure 7.4: The result of transforming the coffee-machine state machine from Fig. 7.2 into a Petri net

QVT-O or ATL—which we actually did in our online repository) that make accessing the models even easier. However, which transformation language you choose is a not an easy decision and is influenced by many factors. Mainly, it should integrate well into your system and it should be a language that allows easy maintenance and evolution. For instance, if in your organization, there is no expert in the language, or the one expert might leave the organization, then it is better to make a pragmatic choice and rather pass on using certain language features, but stick to a mainstream language, such as Java, that is widely used in your organization. See the discussion about technological spaces in Sect. 7.1.

Implementing the transformation. Figure 7.5 shows our Scala implementation. From a first look, you will see that our strategy is to decompose the overall transformation into Scala methods that take care of different model parts. We call those parts rules and in Scala put them into methods. Specifically, we transform states to places with `convertState`, transitions with `convertTransition`, end states to transitions that throw a token away with `convertEndState2RemTrans`, and state machines to Petri nets with `convertStateMachine`. Beyond two other helper methods (`getInitial-TokenCount` and `isEndState`), which query the models and are used in the rules, there is just the method `run` to start the transformation. You will see that there, we start the transformation by transforming state machines (instances of the meta-class `FiniteStateMachine`).

```scala
object FsmToPetriNet extends CopyingTrafo[fsm.Model, petrinet.Model]:
  val pFactory = petrinet.PetrinetFactory.eINSTANCE

  // We inject some queries over States
  extension (self: fsm.State)
    def getInitialTokenCount = if (self.getMachine.getInitial == self) 1 else 0
    def isEndState = self.getLeavingTransitions.isEmpty

  def convertState (self: fsm.State): petrinet.Place =
    pFactory.createPlace before { p =>
      p.setName(self.getName)
      p.setTokenNo(self.getInitialTokenCount)
    }

  def convertTransition (places: List[petrinet.Place]) (self: fsm.Transition)
    : petrinet.Transition =
    pFactory.createTransition before { pnt =>
      pnt.setInput(self.getInput)
      pnt.getFromPlace.addAll(places.filter(_.getName == self.getSource.getName).asJava)
      pnt.getToPlace.addAll(places.filter(_.getName == self.getTarget.getName).asJava)
    }

  def convertEndState2RemTrans (places: List[petrinet.Place]) (self: fsm.State)
    : petrinet.Transition =
    pFactory.createTransition before { pnt =>
      pnt.setInput("")
      pnt.getFromPlace.add(places.find(_.getName == self.getName).get)
  }

  def convertStateMachine (self: fsm.FiniteStateMachine): petrinet.PetriNet =
    pFactory.createPetriNet before { pn =>
      pn.setName(self.getName)
      val places = self.getStates.asScala.toList.map(convertState)
      pn.getPlaces.addAll(places.asJava)
      pn.getTransitions.addAll(
        self.getStates.asScala
        .flatMap(_.getLeavingTransitions.asScala.toList)
        .map(convertTransition(places))
        .asJava)

      // for each end state generate a transition that quashes a token
      pn.getTransitions.addAll (
        self.getStates.asScala
        .filter(_.isEndState)
        .map(convertEndState2RemTrans (places))
        .asJava)
    }

  override def run (self: fsm.Model): petrinet.Model =
    pFactory.createModel before {
      _.getPetrinets.addAll(self.getMachines.asScala.map(convertStateMachine _).asJava) }
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/transforms/FsmToPetriNet.scala

Figure 7.5: *Transformation of* `fsm` *models to* `petrinet` *models in Scala*

Rules call other rules explicitly. So, we explicitly encode control flow. There are dedicated model-transformation languages that do not require this; you just declare rules, and the transformation engine figures out the best way of applying them. In Scala as a GPL, we need to figure out a viable strategy ourselves and encode it explicitly in the transformation code.

Let us look at the different parts of the transformation in detail.

At the top (Scala object declaration), see that our transformation is implementing the trait `CopyingTrafo[fsm.Model, petrinet.Model]`, which is a trait we defined in the book-accompanying libraries (`dsldesign.scala.emf`) that we offer to ease the use of Scala. It just defines the run methods. An alternative trait would be `InPlaceTrafo[T]`, which only takes one meta-model as a type parameter and, as we will discuss below, just modifies the model at hand instead of creating a separate transformed "copy" of the model—as in our example.

Next, the field `pFactory` holds a reference to EMF's object instantiation factory, which is the standard way of instantiating meta-classes defined in Ecore. It is specific to our target meta-model and is part of the generated code from our `petrinet` meta-model defined in Ecore.

Let us now go through the different parts of this transformation, starting at the bottom, where the entry point is, and then gradually going up.

At the very bottom, the method `run` is our entry point. We see that it creates an instance of the meta-class `Model`, which is the top class in the partonomy of the meta-model and contains `PetriNet` classes. So, we need to instantiate `Model` first, which will hold instances of the rest of the target `petrinet` model. Before we return the newly created `Model` instance as a result of the transformation, we need to set its attributes, of which there is only one in this case: the list of Petri nets. To write this initialization more concisely we created in the book-accompanying library `dsldesign.scala` an operator `before`, which allows us to avoid introducing local name bindings.[5] So, instead of instantiating `Model`, assigning it to a local `val`, and then setting the attributes, we can avoid that by giving a closure to `before`, which will evaluate the closure and then return the instance. In the closure, we can just run a number of statements to fill the attributes. Observe the other uses of `before` in our transformation. However, it only saves two lines (the `val` assignment for the name binding, and the return of the instance), so feel free to just use local name bindings. The actual conversion is then done using Scala's `map` function, which runs `convertStateMachine` on all state machines contained in the attribute `machines` of `Model`. Note that we, somewhat awkwardly, need to wrap the mapping into `.asScala` and `.asJava` calls.

Next, the method `convertStateMachine` works as follows. It uses the operator `before` to offer a reference to the newly created instance of a `PetriNet`, whose attributes we now set:

- We first set `name` to the name of the state machine we convert from.
- Then, we convert all states to places. Interestingly, we add a name binding here and hold the newly created places in a `val places`. The

[5]This operator is inspired by the operator `=>` in the transformation language Xtend.

reason is that we will need to query our mapping of states to places later again. Here, we actually see a disadvantage of using a GPL instead of a dedicated model-transformation language, such as QVT-O. The latter contains a 'trace model' that collects and holds the mapping in memory, so that one can query it retroactively. Holding the collection of places is our way of emulating a trace model. You will see how we query it in `convertTransition` and in `convertEndState2RemTrans` below. We then, of course, also add the places to the respective attribute (`places`) in our `PetriNet` instance just created.

- Thereafter, we map transitions. Notice the different structures, that is, the partonomies, in the meta-models. In the `fsm` meta-model (Fig. 7.1), transitions are contained by states, whereas in `petrinet` (Fig. 7.3), the transitions are contained by the meta-class `PetriNet`—this structural difference complicates our transformation slightly and requires the querying of our emulated 'trace model' (the list `places`). To actually transform the transitions, we obtain the list of them by collecting them via the states and then flattening the list, and then mapping them to Petri net transitions via `convertTransition`, passing it the `places` list.
- Finally, we need to create the pendant of state-machine end states in the `petrinet` model, which are dangling transitions (i.e., they quash tokens). To this end, we obtain all end states by filtering the list of states using the query `isEndState` we created as an extension method for the class `State`[6] in the upper part of the transformation. We then map those end states to dangling Petri net transitions using `convertEndState2RemTrans`.

The method `convertEndState2RemTrans` does the conversion we just discussed. It sets the newly created transition's attribute `input` to an empty string (which means that the transition always fires and then destroys the token, since there is no other place, so that it is the equivalent to an end state in a finite-state machine). It then maps the transition's `fromPlace` attribute to the place that corresponds to the end state. Note that here, it is important that we retrieve the previously created instance of a place from our emulated 'trace model' (the `places` list) and do not create a new instance! In dedicated model-transformation languages (e.g., QVT-O), we would not need to pass such a list of already mapped instances, but could just query the real trace model for the place instance that the respective end state was mapped to in another transformation rule.

The method `convertTransition` converts the transitions and also needs to look up the places corresponded to by the start and end attributes of the state-machine transitions that we convert. Specifically, we instantiate a new Petri net transition, set its attribute `input` to the `input` of the state, and then set the attributes `fromPlace` and `toPlace` by looking up the place instances from our 'trace model' (which we passed as an argument to the method)

[6]Scala extension methods allow adding methods to types that are already defined. Here, recall that the type `State` was defined in the meta-model and materialized in the Java classes generated from the meta-model.

that correspond to the respective source and target states of the original state-machine transition.

At the top, the method `convertState` is probably the simplest mapping rule. It creates a new `Place`, sets the `name` attribute to that of the source state, and sets the `tokenNo` (number of tokens in the place) via the query method `getInitialTokenCount` defined as an extension of the class `State`.

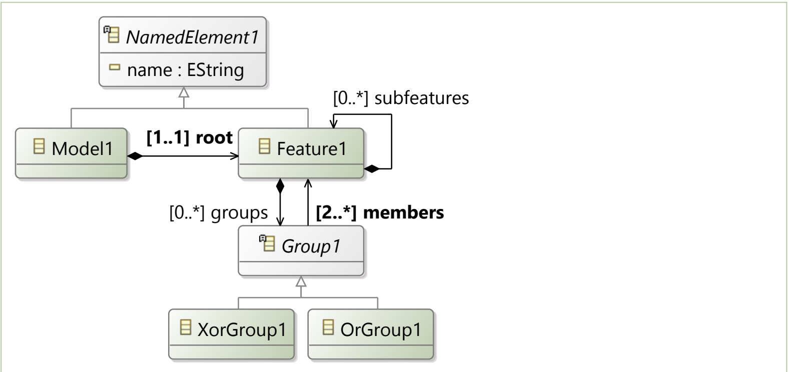
Comparison Across Model-Transformation Languages

To illustrate what our transformation from `fsm` to `petrinet` looks like in different languages, including the dedicated model-transformation languages QVT-O, ATL, and Xtend, Tbl. 7.1 repeats the transformation rule `convertTransition` from our first case study (Sect. 7.2) and compares it with the other languages.

Case Study 2: Transforming Feature Models

A common use case for model-to-model transformations is to translate between two similar languages (for example, between two versions of the same tool). Imagine an organization has shipped a modeling tool—here, we will consider feature models again. Users of the tool have created models in the language. Now, for some optimization reasons, the organization changes the language. Any update to the tool shipped to the customers will need to convert models in the old language into the new version of it.

We discuss such a transformation, but more concisely than the one above, omitting various details about using Scala to implement a transformation. So, you should have read Sect. 7.2 before reading this section.



source: featuremodels/model/FeatureModels1.ecore

Figure 7.6: A meta-model for feature diagrams

Transformation goal. Our goal is to transform feature-model instances represented in one meta-model into instances of a slightly modified meta-model for feature models.

Recall two meta-models for feature models presented in Chapter 3. We repeat them here in Figures 7.6 and 7.7. In the first meta-model, observe that sub-features are contained in the `subfeatures` collection of the parent

Scala: fsm.scala/src/main/scala/dsldesign/fsm/scala/transforms/FsmToPetriNet.scala

```
def convertTransition (places: List[petrinet.Place]) (self: fsm.Transition)
  : petrinet.Transition = pFactory.createTransition before { pnt =>
    pnt.setInput(self.getInput)
    pnt.getFromPlace.addAll (places.filter (_.getName == self.getSource.getName).asJava)
    pnt.getToPlace.addAll (places.filter (_.getName == self.getTarget.getName).asJava) }
```

We use the operator `before` (implemented in the book's library) to assign attributes to the newly created `Transition` instance without creating a name binding (e.g., `val newTransition = pFactory.createTransition`). Since we need to set the `fromPlace` and `toPlace` attributes to already instantiated `Place` objects, we carry our self-made trace model in terms of a list `places: List[petrinet.Place]` and retrieve objects from there. After filtering that list, we use `asJava` to convert it to a Java collection used in the EMF API, so that we can add it to the Ecore model.

QVT-O: fsm.qvto/transforms/ToPetriNet.qvto

```
mapping FSM::Transition::ConvertTransition(): PN::Transition{
  input := self.input;
  fromPlace := self.source.resolveone( PN::Place );
  toPlace := self.target.resolveone( PN::Place ); }
```

QVT-O is also imperative, so the logic is similar to our Scala implementation, but we save the explicit object instantiation as well as maintaining and carrying over our own trace model. Instead, QVT-O provides methods to retrieve already instantiated target objects by querying for source objects, or vice versa, from its trace model. Here, we get the `Place` object that was created for the `State` object from the original transition's `source` attribute; similarly for `toPlace`. One can also retrieve multiple objects for associations with a * cardinality, or retrieve only those that were transformed by a specific rule. Details on QVT-O are in the books of Gronback [32] and (in German) Nolte [53].

ATL: fsm.atl/transforms/ToPetriNet.atl

```
rule Transition2Transition{
  from t: FSM!Transition
  to tr: PN!Transition(
    input<-t.input,
    fromPlace<-thisModule.resolveTemp(t.source, 'p'),
    toPlace<-thisModule.resolveTemp(t.target,'p') ) }
```

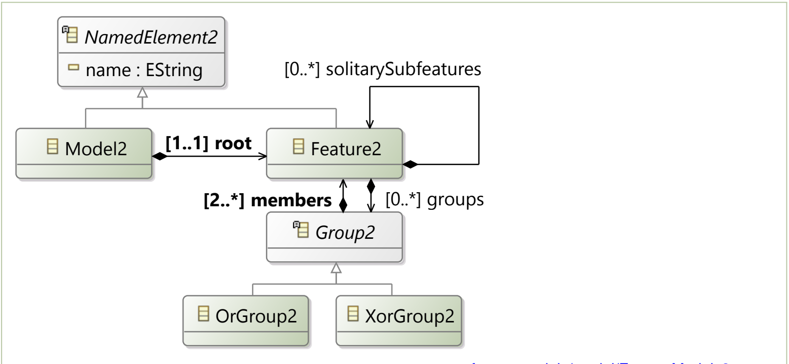
ATL is imperative and declarative. Here we use the declarative style, where one defines a rule, and whenever the source pattern (after `from t:`) matches in the source model, the rule is applied. The rule is not explicitly called from another rule, as opposed to Scala, QVT-O, and Xtend (where we call it in the rule that transforms state-machine objects to Petri net objects), which makes ATL concise. Like QVT-O, we do not need to explicitly instantiate the target object, nor to maintain and carry over our own trace model. We access the trace model with the `resolveTemp` method provided by the ATL API, but we need to specify the name of the target pattern (`'p'`), which is defined in some other matched rule. Detailed documentation about ATL is available on its website: https://www.eclipse.org/atl/documentation.

Xtend: fsm.xtend/src/main/xtend/dsldesign/fsm/xtend/ToPetriNet.xtend

```
def dsldesign.petrinet.Transition convertTransition( Transition t, EList<Place> places ){
  pFactory.createTransition => [
    input = t.input
    fromPlace += places.filter[ x | x.name == t.source.name ]
    toPlace += places.filter[ x | x.name == t.target.name ] ] }
```

Very similar to the Scala implementation. We use Xtend's `=>` operator to avoid a local name binding (e.g., `val pntransition = pFactory.createTransition`) for the explicitly instantiated Petri net transition object. Otherwise, we also maintain and carry over our own 'trace model' and query it for the already instantiated `Place` objects. We named the parameter `t` instead of `self`, which is a reserved keyword. Like QVT-O and ATL, Xtend is nicely integrated with EMF and uses its collection API, so there is no need to convert from e.g., `EList` to another collection type, in contrast to Scala.

Table 7.1: The rule that transforms transitions of a state machine model implemented in four model-transformation languages. The full Scala implementation is in Fig. 7.5, the others are in the book's repository



source: featuremodels/model/FeatureModels2.ecore

Figure 7.7: *An alternative meta-model for feature models*

feature. Then, if these features are part of a group, an object of type `Group1` is placed under the feature object with references to the features that are group members. In the second, alternative meta-model for feature models, which is perhaps simpler, study the relation between classes `Feature2` and `Group2`. This time only solitary features are nested directly under the parent feature. Grouped features are contained in an object of class `Group2`. Compare to the first meta-model, where only a simple reference was used for `members`. The advantage of the first meta-model was that it was easy to access all sub-features—now, we need to combine the set of solitary sub-features with the members of all groups to compute the set of all children. On the other hand, in this new meta-model we do not need to write two important constraints that are already guaranteed by types: (i) in an instance of the first meta-model, a group should only contain sub-features of its parent (and not of other features), and (ii) any two groups nested under the same feature should not overlap (they should have disjoint sets of members). Remember Exercise 5.40, in Chapter 5, where we asked you to express these two constraints.

Transformation requirements. Transforming models (i.e., instances) of FeatureModels1 into models of FeatureModels2 requires mainly copying the objects and then creating a slightly different partonomy. This transformation is probably simpler than the one above (Sect. 7.2), but the difference in partonomy might be a bit tricky. Essentially, the requirements are to copy objects of type `Model1`, `Feature1`, and `Group1` (with its sub-classes `XorGroup1` and `OrGroup1`) into respective objects of type `Model2`, and so on. When copying the features, specifically when creating the feature hierarchy in the target model by filling the `solitarySubFeatures` composition relation, only those features should be added that are not part of a group. Instead, these should be added to the members relation, which is a composition relation in the target model instead of just an association.

Transformation technology. We implement the transformation again in four languages: Scala, QVT-O, ATL, and Xtend. We discuss only the Scala one here, and refer to our online repository: featuremodels.xtend, featuremodels.qvto, featuremodels.atl, and featuremodels.scala for the full implementations.

Transformation implementation. If the transformation logic that you should implement is not clear from the requirements, you should take a look at example models in abstract syntax. Recall Exercise 3.17 and Exercise 3.18 from Chapter 3, where we already asked you to create such instances for these two meta-models.

Figure 7.8 shows our Scala implementation. Similarly to our transformation of `fsm` into `petrinet` models above (Sect. 7.2), we decompose the transformation into methods, each of which handles the transformation of a meta-class. Let us go through it from bottom to top.

The method `run` is the entry point and delegates the conversion of a model instance to the method `convertModel`.

The method `convertModel` creates a new model instance using the EMF object creation factory that is generated from the meta-models (see the Gradle build script for details, or alternatively the appendix "Using the Eclipse Modeling Framework" on our book website http://dsl.design). It uses the `before` operator, in whose body we bind the reference `m` to use it twice: to set the name of the new `Model2` instance to that of the source model, and to set the `root` containment relation to a converted root feature.

Then, `convertFeature` instantiates a new feature, sets the name to that of the source feature, and then fills the `solitarySubFeatures` composition relation as described in our requirements above. To figure out whether a sub-feature is solitary or a member of a feature group, we defined the query `isSolitary` as an extension method to the class `Feature1` at the top of our transformation implementation. Thereafter, the other features (i.e., those that are part of a group) will be added as children to a new `Group2` instance in `convertGroup`.

Finally, `convertGroup` is responsible for creating the right `Group2` instance (either `OrGroup2` or `XorGroup2`) and then converts and adds all grouped features (which are found via the `members` association in the source model) to the target model, where they are contained in the `members` composition relation, so they become children of the group in the target model, whereas they have been children of the feature in the source model.

Exercise 7.1. Now consider yet another alternative meta-model for feature models in Fig. 7.9, which should be the new target meta-model of the transformation. In this meta-model, there is no longer a type distinction between OR and XOR groups. Instead, groups get a lower and upper bound specifying the number of features that can be selected within them. OR groups map to a lower bound of 1 and no upper bound (represented by a '*' or '-1'), while XOR groups map to a lower bound of 1 and an upper bound of 1.

Take the existing transformation and change it so that it works correctly with the changed target meta-model. You can also do this exercise on paper by crossing out, adding, and changing any part of the transformation in Fig. 7.8 as you consider it meaningful.

```scala
object FeatureModel1ToFeatureModel2 extends CopyingTrafo[Model1, Model2]:

  val fm2Factory = featuremodels2.Featuremodels2Factory.eINSTANCE

  extension (self:Feature1)
    def isSolitary(subfeature: Feature1): Boolean =
      !self.getGroups.asScala
        .exists {_.getMembers.asScala.exists(_ == subfeature)}

  def convertGroup (self: Group1): Group2 =
    { if (self.isInstanceOf[OrGroup1])
        fm2Factory.createOrGroup2
      else
        fm2Factory.createXorGroup2
    } before {
      _.getMembers.addAll(self.getMembers.asScala.map(convertFeature).asJava)
    }

  def convertFeature (self: Feature1): Feature2 =
    fm2Factory.createFeature2 before { f =>
      f.setName (self.getName)
      f.getSolitarySubfeatures.addAll (
        self.getSubfeatures.asScala
        .filter (self.isSolitary)
        .map (convertFeature).asJava)
      f.getGroups.addAll (self.getGroups.asScala.map (convertGroup).asJava)
  }

  def convertModel (self: Model1): Model2 =
    fm2Factory.createModel2 before { m =>
      m.setName(self.getName)
      m.setRoot(convertFeature(self.getRoot))
    }

  override def run (self: Model1): Model2 =
    convertModel (self)
```

source: featuremodels.scala/src/main/scala/dsldesign/featuremodels1/scala/transforms/FeatureModel1ToFeatureModel2.scala

Figure 7.8: *Transformation of feature models between two different meta-models (Figures 7.6 and 7.7)*

Summary

Both transformations are relatively simple, especially the feature-model one. Still, the latter shows that already a seemingly small, but tricky mismatch between the source and target meta-model can require you to write such upgrade transformations and ship them with a new version of your tool.

We implemented both transformations in Scala, but our repository contains the full implementations also in QVT-O, ATL, and Xtend. In principle, one could have written the transformation also in Java, which would probably not even be so much more verbose, since Java also offers functional collection operators, such as map and filter, which take closures (anonymous functions) as in Scala as parameters. One could have avoided the explicit conversions between Java and Scala lists, since the code generated from the Ecore meta-models is in Java.

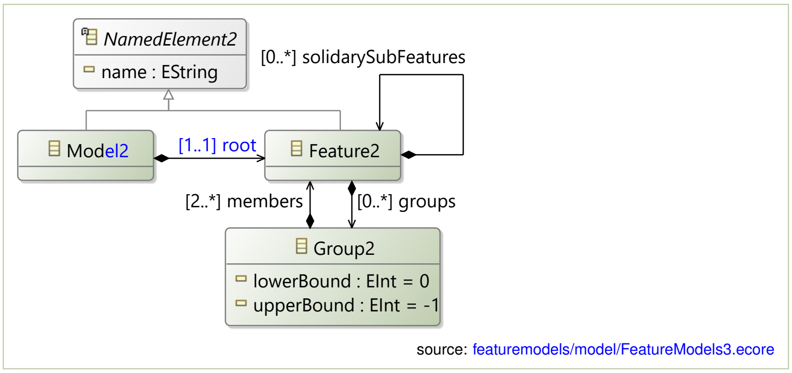


Figure 7.9: *Another alternative meta-model for feature models (changed target meta-model)*

Another observation is that in Scala, we need to explicitly encode control flow instead of declaring transformation rules and then letting the transformation figure out how and in what order to call them and how to traverse the source model (as the transformation language ATL would do). In the signatures of the rules, we explicitly wrote down the return types, which could be omitted in Scala. However, we left them in for better comprehension, especially here in the book.

7.3 Applications of Model and Program Transformation

Model- and program-transformation techniques have many applications. In the context of building DSLs, transformations are mainly used to realize the dynamic semantics by translating models into other models. However, they can also support the management of models. In a way, transformations then also implement the semantics of languages, perhaps not completely and only focusing on a specific aspect, so it is fair to say that the main application of transformations is to capture language semantics in one form or another.

Beyond the examples we show in this book, there are repositories of model transformations. The ATL transformation zoo[7] contains more than 100 documented transformations. Kusel et al. [42] and Selim, Cordy, and Dingel [60] survey and study these transformations. Whether one needs a dedicated transformation language for rather small transformations is a question we discuss in Sect. 7.9, but the transformations in the zoo provide an overview of typical problems addressable by model or program transformations, and especially how such transformations can be structured and decomposed into individual rules.

Dynamic Semantics Definition

Back in Chapter 2 we already talked about the term dynamic semantics. Let us, however, recall the distinction between syntax and semantics of a language. The term syntax refers to “the principles and processes by which sentences are constructed in particular languages” [15]. It is the

[7] https://www.eclipse.org/atl/atlTransformations, retrieved 2022/09

actual representation of a model that users or tools interact with. The term semantics refers to the meaning or the effect of a model.

Semantics can be defined informally or formally. In the former, you basically put (and hide) the meaning of models in source code, which has some behavior influenced by a model. You can always do that, but by expressing semantics more formally, you make the semantics available to automation and analysis. For instance, you can identify certain properties of the semantics, such as correctness, liveness (e.g., no deadlocks), or safety. Interpreters, which we will discuss in Chapter 8, can be seen as a more informal way of specifying dynamic semantics. Still, when implementing the interpreter systematically, following patterns and best practices, you might be able to analyze the semantics' properties as well to some extent.

For formal semantics [50, 72], we distinguish between different styles, for example:

- *Operational semantics* define the meaning of a language in terms of states that the instances can have, together with defining valid transitions among the states. The transitions capture the possible sequences of states. A state is an abstraction of one possible state of the whole model. If you have a simple programming language that allows definition of variables, then a state typically represents a mapping of variables to values at a given execution step. All the states in the operational semantics of a program are typically derived from the possible combinations of variables and values. However, from this example, you sense that operational semantics are rather uncommon in MDSE.
- *Denotational semantics* define the meaning of a language inductively, by declaring denotations for smaller parts of the language, which when combined in some form, determine the whole semantics. As such, the main property of denotational semantics is compositionality. Denotational semantics are well-suited to describe, for instance, the semantics of feature-modeling languages or other variability-modeling languages, such as Kconfig from the Linux kernel, which we will discuss in Sect. 11.2. We refer to technical notes we created that show what denotational semantics for variability-modeling languages look like: She and Berger [61] and Berger and She [5].
- *Translational semantics* define the meaning of a language by specifying how instances in one language are transformed into instances of another language, for which the semantics is already implemented or well known. This kind of semantics is most common in MDSE: all our transformation examples in this chapter realize a kind of translational semantics.

Let us look at two examples.

Example 17. Recall some of the examples of semantics we discussed in Chapter 2 (Example 5 and Sect. 2.4), where we had a very simple state-machine-like DSL called the `robot` control language. In this case, a suitable semantics of a model was the set of all possible execution traces (i.e., sequences of modes, actions, and events) that are allowed at runtime.

Example 18. Recall our `fsm` language from Example 6 (Chapter 3). Depending on the purpose, finite-state machines can have different semantics, for example:

- In Example 6 we mentioned that it should be used for teaching purposes, where students execute state machines and observe the behavior. Here, it makes sense to translate the state machine into source code in a GPL, such as Java. The source code incorporates the switch pattern (illustrated in Fig. 9.8 on p. 332) or the state pattern [29] to structure the states and their transitions. We will show this prime example of a translational semantics in Sect. 9.4 (see Figures 9.8 and 9.9). Another alternative is to translate the state-machine model into a data structure that is evaluated at runtime by an interpreter [56]. This shows that different types of semantics can also be combined.
- Another common purpose of state machines is to define acceptable input words (or sentences). In this case, the semantics we are interested in is the set of all possible words (or sentences) over an alphabet of input characters (or input words). You can define these semantics by transforming state machine models into regular expressions, whose semantics are known, so we have another common example of a translational semantics here. You can also write an interpreter that checks for a given input word (or sentence) whether it is acceptable (by just traversing the states via the transitions, which are labeled with inputs). This also defines a semantics, but less formally, since it is hidden in the implementation of the interpreter.

Model Refinement

Translating models or programs is the most important application of model- or program-transformation technologies. And, as we just discussed, translating models is a common way to define the dynamic semantics of DSLs.

Generating lower-level models from models at a higher level of abstraction [21] is most typical. This kind of transformation is usually called *refinement* [8]. You often see refinement translations chained one after another, typically ending with the generation of source code, which is at the lowest level of abstraction.

Example 19. A good example is expanding syntactic sugar. Languages incorporate syntactic sugar to optimize the language and ease its usage, which leads to more concise and more comprehensible models. Back in Sect. 4.2, specifically in Tbl. 4.1, we described syntactic sugar for regular expressions. When you have syntactic sugar in your language, it makes sense to 'desugar' the respective elements in your model and convert them into non-sugar elements, since then you do not need to implement semantics for the syntactic sugar as well, avoiding redundancy. Often, expanding syntactic sugar happens conveniently in the parser on the concrete-syntax level. But, when you do not use a parser, or believe that the parser specification is already complex enough, you can realize desugaring as a refactoring transformation which operates on the abstract syntax.

Model Abstraction

The opposite of model refinement is model abstraction, which refers to generating higher-level models from lower-level ones. An example would be *reverse-engineering* of models, such as extracting feature models from source code.

Example 20. A more concrete example would be translating simulation models (e.g., of a flight simulator [46]) between different levels of fidelity. High-fidelity models are more detailed. For instance, a transformation could abstract a time given in seconds (high fidelity) into the values fast or slow (low fidelity), so it translates the lower-level model into a higher-level one that is an abstraction.

Model Management

When you use the DSL(s) you developed in your software engineering projects—that is, when you adopt MDSE—you will need to manage models and related artifacts. You will need to evolve and maintain them. While refinement and abstraction already help in managing models, the following activities specifically focus on model management. They can be seen as rather internal MDSE problems that are mainly addressed by the vendors of MDSE tools.

- *Model Mapping* and *Model Synchronization* of models at the same level or different levels of abstraction. For our flight simulator example, it is necessary to also maintain models at different levels of fidelity [46], which requires synchronization or alignment.
- *Model View Creation* comprises querying models for information and establishing views on models. When changing the view, ideally, transformations can incrementally update the original model as well. As an example for views, consider our feature-model transformation from Sect. 7.2. From feature models, a query could extract all grouped features, or it could provide a slice through the feature hierarchy presented as a feature-model view [37] to the user.
- *Model Evolution* tasks include model versioning, model comparison (a.k.a. diffing), model merging, model update (including incremental update), and patching. Furthermore, languages evolve, so their instances might need to be updated to the new language version. Model evolution also includes migration [32] from one language to another language at the same level of abstraction (changes in that level are captured by *Refinement* and *Abstraction* above).

Model Optimization

Models sometimes need to be optimized with respect to some property of their syntax or semantics. For instance, syntax-related properties are the model size, structure, comprehensibility, or conciseness, which can be optimized. Semantics-related properties can be memory consumption or performance when models are executed. Optimizations are often semantics-preserving and are then called *refactorings* [32], but they can also change the semantics when reasonable.

Model optimization has always been the main objective of program transformation. The idea is to transform a program so that it is optimized towards a specific property. To this end, program transformations manipulate programs or program parts, typically within compilers, refactoring engines, or program analyzers.

For example, our transformations in Sect. 7.5 show the manipulation of Boolean expressions with the goals of reducing their size by removing redundancies, enhancing comprehension, or making expressions compatible with programs (SAT solvers) that need expressions in a certain format (conjunctive normal form). Let us look at some further examples.

Example 21. A typical application mentioned in the literature is constant folding [55, 9]. In our expression example, you basically replace sub-expressions that are constant. For instance, when identifiers have a value, then you replace the identifier with the value, which simplifies the expression. Applied to programs, constant folding is a typical program optimization technique that focuses on optimizing data flow in programs.

Example 22. Other common applications are copy propagation and dead-code elimination in programs. For instance, when an IF statement's condition always evaluates to false, dead-code elimination removes the true branch and just leaves the false branch. Just to name a few more, further program optimizations are: fusion, inlining, constant folding, common sub-expression elimination, or partial evaluation (a.k.a. program specialization). There is a whole book about partial evaluation by Jones, Gomard, and Sestoft [38].

In summary, the goal of applying transformations for optimizations is most often not to change the semantics, but to change other properties. This is obviously also the case when we use transformations to define the dynamic semantics of DSLs. For the other applications—translating models to lower levels of abstraction and managing models—one usually changes the semantics.

7.4 Transformation Fundamentals

In general, model transformations are specialized programs that take source models as input and either modify them or produce new target models as output. They traverse the source models and copy, modify, or create model fragments. The transformations are defined in a language, which can be a GPL or a dedicated transformation language, which can be in the form of a library. Let us present some basic definitions.

Definition 7.2. *A* model transformation *is a computable function that maps a set of source models to a set of target models. It comprises one or multiple transformation rules.*

A model transformation establishes a relationship between one or several source models and one or several target models. This relationship is computable, at least in the direction from the source to the target model.

Transformation Architecture

Figure 7.10 shows an overview of a model transformation with architectural relations to artifacts in a model-driven software project. In the middle is the actual transformation definition—the code specifying the transformation rules (explained shortly). It is executed by a transformation engine, which is either the runtime environment of a specialized model- or program-transformation language (e.g., QVT), or the execution environment of the GPL in which the transformation is defined (i.e., Scala in our examples).

Definition 7.3. *A* transformation engine *is a tool that executes a transformation definition on source models and produces target models.*

Specialized transformation engines often have additional capabilities to scale and maintain model transformations, including parallel execution and model traceability management.

The source and target meta-models are typically used to type-check the implementation of the transformation, as languages used to implement transformations tend to be statically and strongly typed (although not necessarily so). The transformation is executed by a transformation engine (mid-bottom in the figure), which, while interpreting the transformation code, reads the source model, and produces the target output. The transformation itself is implemented in (i.e., conforms to) a suitable transformation language, either a model-transformation DSL (such as QVT) or a suitably powerful GPL (for instance, Scala supported by a library such as Kiama).

Note that, while the transformation transforms source models into target models, the transformation is defined over the source and target meta-models. In other words, the transformation definition refers to elements (meta-classes) of the source and target meta-models. In the figure (Fig. 7.10), we show Ecore as the meta-modeling language, but of course, the meta-models can be realized using algebraic data types as well. This is the typical case when using program-transformation technology.

Model transformations are often chained. You might immediately remember a compiler, where you have different transformations chained one after another to generate lower-level code. Similarly, when one creates

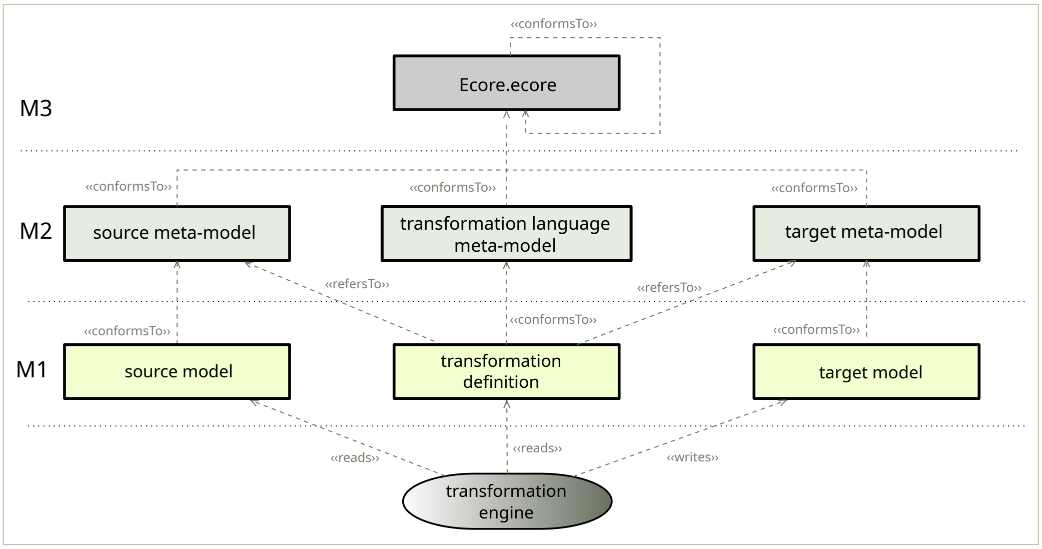


Figure 7.10: A typical model-transformation architecture shown in the meta-modeling hierarchy

code from a higher-level language, such as a DSL, one often has multiple transformations.

Transformation Rules and Rule Application

While a large number of model- and program-transformation techniques have been presented, they all provide one common abstraction: the *transformation rule*. According to Visser [70], a "rule defines a basic step in the transformation."

Definition 7.4. *A* transformation rule *is a function mapping fragments of a source model to fragments of a target model.*

Transformation rules define patterns over fragments in the source and target models. They also explicitly define what fragments of the source model they are applicable to. Rules can usually access context information, such as trace models or transformation parameters.

Rules are selected and applied to the source model in a certain order (or in parallel), which is controlled by the transformation engine or by the developer. In all our examples in this chapter, the rule application is determined by the Scala runtime, and we as developers control the application of rules to some extent. For instance, we call rules explicitly from other rules, we define so-called rewriting strategies explicitly (an abstraction of controlling traversal and rule application, discussed in Sect. 7.6), or we let Scala decide which rule to apply via pattern matching.

Classifications

Various classifications of model transformations exist in scientific articles [48, 21, 39] and in books [8, 17]. The following are the most common categorizations of model transformations.

M2M versus M2T Transformations Model transformations are separated into model-to-model (M2M) and model-to-text (M2T) transformations. These mainly differ in the kind of output generated.

M2M transformations convert models in abstract syntax into models in abstract syntax, each of which adheres to a meta-model. Working on the abstract syntax (cf. Sect. 2.4) has the advantage that there is no irrelevant information from the concrete syntax, which usually just complicates a transformation.

M2T transformations convert models directly into text—that is, abstract syntax into concrete textual syntax (essentially strings). The latter is sometimes also called pretty-printing, especially in the context of program transformations. In M2T, the output meta-model is trivial, since you output a (potentially large) string of characters.

M2T transformations usually offer less type safety compared to M2M transformations, where the target models adhere to dedicated meta-models. One would need to check the generated textual output using a grammar or a regular expression, for instance. An alternative is statically typed macro languages, which work similarly to macros in the C preprocessor, but are statically typed and cannot create an invalid AST. Such macros are available in Scala as part of its "principled meta-programming" capabilities.

When we chain transformations into larger ones, we usually use M2M transformations except for the last one, which is often the transformation into text (M2T).

Finally, you might sometimes even see the term text-to-model (T2M) transformations. That most often just refers to parsing, so transforming a model from concrete syntax into abstract syntax. It might involve changing the semantics or translating into another language; then it is not exactly parsing anymore. T2M is usually at the beginning of a transformation chain. Furthermore, there are transformation systems that work on parse trees or on plain texts, so could be dubbed text-to-text (T2T) transformations, but they are rather uncommon in our experience.

Horizontal versus Vertical Transformations We also classify transformations based on whether and how the abstraction level among source and target models is changed. Figure 7.11 illustrates these kinds, referring to applications from Sect. 7.3 above. When the level is changed—for instance, when realizing refinement or abstraction with the transformation—we talk about a *vertical transformation*. When we only change the structure of the model—for instance, when realizing evolution or optimization—we talk about a *horizontal transformation*. A transformation can also change both; then we informally call it a *skewed transformation*.

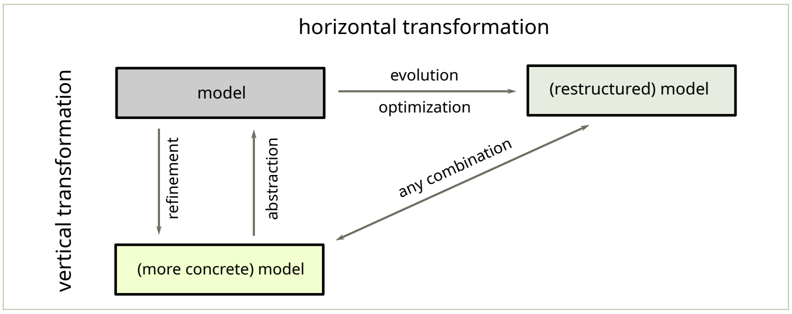


Figure 7.11: *Horizontal versus vertical transformations*

Endogenous versus Exogenous Transformations Another typical characterization of transformations is based on whether the source and target models conform to the same language or not—in other words, whether the meta-models are the same or not.

Definition 7.5. *An* endogenous *transformation (endo-transformation) translates instances of a language to instances of the same language.*

Endogenous transformations are also known as *rephrasings* [70]. A trivial one is the identity (no change to the source model). Model optimization transformations are often endogeneous, but (cf. Sect. 7.3) refactoring (changing the models without changing semantics), desugaring (implementing more complex features as less complex ones), and constant folding (replacing constant variables in programs) are always endogenous.

Definition 7.6. *An* exogenous *transformation translates instances of one language into instances of another language.*

Exogenous transformations are also just called *translations* [70]. The transformation from `fsm` to `petrinet` is *exogenous*. Other common exogenous transformations are those that realize translational semantics, such as compiling source code and models to intermediate representations, visualizing models in HTML or SVG, and translating models for analysis using solvers (e.g., our translation of logical expressions into conjunctive normal form below in Sect. 7.5).

Deciding whether a transformation is endogenous or exogenous is simple for M2M transformations. It is based on the equality of meta-models. For M2T transformations, or chained transformations that start with a textual source model, end with a textual target model, or both, it is not so clear. To this end, we define language equality over the abstract syntax here. So, if the source and target models conform to the same meta-model (even if not explicitly provided), we call the transformation endogenous, and exogenous otherwise. But, since the concrete syntax can be different, the models might look like they belong to different languages.

In-Place versus Copying Transformation To produce a target model, a transformation can take the source model and modify exactly that model, or it can copy and modify (parts of) the source model.

Definition 7.7. *An* in-place transformation *(a.k.a. destructive transformation) is an endogenous transformation that modifies the source model, instead of producing a new model.*

In-place transformations are practical for small model adaptations, such as normalization of model element names, or for refactorings, such as variable renaming. They are also very useful for models used in a model-view-controller pattern, where the controller modifies a model directly, and asks the view to update. However, they often are hard to manage for complex manipulations of models. In-place transformations can only be realized in an imperative language, where destructive updates are allowed. In-place transformations only make sense in the endogenous case.

Definition 7.8. *A* copying transformation *(a.k.a. out-of-place or pure transformation) creates a new target model based on the source model.*

One strong advantage of copying transformations is that the source model is available unmodified throughout the process, so we can always refer in rules to the original state. In in-place, destructive transformations, information is often lost during transforming (unless stored separately). A copying transformation is also more testable, as one can write assertions relating the source and target. For destructive transformations, this can still be done, but requires cloning source instances, or referring to their serialized, unmodified original copies.

Rewriting is an example of an endogenous copying transformation. Our transformations in Sect. 7.5 below are of this kind. In the rewriting paradigm, a new model is created by creating incrementally different copies of it using rewrite rules. All exogenous transformations are necessarily copying, as we cannot put instances of the new target model in place of the instances of the source model, as this would violate type correctness.

Unidirectional versus Bidirectional Transformations Most model transformations are *unidirectional transformations*, which means that they are only executable in one direction. All our example transformations in this chapter are of this kind. Obtaining a source model from a target model—that is, running the transformation in the opposite direction—requires writing a new transformation.

However, there exists a certain kind of transformation, called a *bidirectional transformation*, which can be executed in both directions, including obtaining a source model from a target model. This scenario is useful when target models can be updated and the source model should be updated as well, or any potential impact on the source model is of interest. However, bidirectional transformations are less common and require the transformation to be defined in a more declarative style.

Number of Source and Target Models Finally, model transformations can be classified by their numbers of source and target models. Beyond the typical case of one source and one target model, if we have two or more source models, then the transformation is usually called a model merge. A special case is when one model represents a configuration or parameterization (e.g., mainly containing key-value pairs); then we call the transformation *parameterized*. If there is no target model, we call the transformation a model analysis or a model query.

7.5 Program-Transformation Case Studies

After some theory, it is time again to look at some case studies, but this time some that are known from the field of program transformation.

We will now realize transformations by relying directly on programming-language capabilities for defining source and target models, and on transformation libraries for implementing the actual transformation rules. So, we will create meta-models using algebraic data types (ADTs) and transform their instances. While various transformation libraries for programming languages exist, we will use the Scala-based library Kiama [62], which realizes the paradigm of strategic programming [44, 45]. The latter decouples model traversal from rewriting and abstracts both into the notion of *strategy*. Essentially, you define and compose strategies for realizing your model transformation, where Kiama already offers a set of basic strategies. This controls how the *transformation (a.k.a. rewriting) rules* are applied.

The following three case studies are about transforming logical expressions. For simplicity, we limit ourselves to propositional (Boolean) logic. It is quite common that you will need to transform tree-based data. Since expressions are represented as trees, we hope you will find the following examples of transformations useful.

The transformations are inspired by our previous experience of developing analysis techniques for variable source code [6, 52, 7]. What variable code is we will discuss in more detail later (Chapters 11 to 13). What we often needed to do is to transform tree-based data, such as abstract-syntax trees (ASTs) of source code, and we often needed to transform logical expressions, for instance, abstracting arithmetic expressions into (less-expressive) Boolean expressions, transforming Boolean expressions into conjunctive normal form (which is needed for using SAT solvers), simplifying expressions, or just conjoining and disjoining expressions to facilitate our code analysis. On a final note, in a recent work where we were limited to writing code in plain C, creating these transformations was much more cumbersome [28]. So, we strongly recommend using modern transformation technology as presented in this chapter.

Case Study 1: Constant Propagation in Logical Expressions

A very common transformation on logical expressions is constant propagation. Essentially, you have an expression over variables. You set one or multiple variables to a value and want to derive an expression where those variables are replaced with the respective values.

We create ADTs for representing Boolean expressions with a minimum of operators (AND, OR, and NOT). These are shown in Fig. 7.12. You may recall that we already introduced an expression language in Sect. 6.1 (specifically, as part of Fig. 6.2), but that expression language focused on arithmetic expressions. So, for simplicity, we introduce a new one, which we will use again later (Sect. 13.1, specifically Fig. 13.3). A visualization of these algebraic data types is shown in Fig. 7.13.

Let us briefly explain these algebraic data types, which we realized as case classes in Scala:

- The abstract class `Expression` is the super-type for any node in the Expression tree structure. It is sealed, which as you may recall from Sect. 3.5 limits the possible sub-types to the sub-classes defined in the same file. It has some convenience methods for composing expressions, including `&`, which takes another expression and conjoins it with the present one. Obviously, if the other expression is *true*, the method returns the present one. The logic is similar for the other convenience methods.
- The classes `BinaryExpression` and `UnaryExpression` represent the two kinds of intermediate nodes to establish the tree structure of expressions. A binary expression node has two children (attributes `left` and `right`), and a unary expression only has one child (`expr`).
- The case classes `AND`, `OR`, `NOT`, and `Identifier` then represent the actual binary and unary expression nodes that we can use for instantiating expressions. They contain a simple pretty-printer via `toString`.
- The case objects `True` and `False` represent Boolean constants together with convenience methods for conjoining, disjoining, and negating them with other expressions. The pretty-printer is also built-in via `toString`. We could have defined them as case classes as well, which would be almost as convenient, since one can easily instantiate case classes in Scala, but this way we share the instances for `True` and `False` expression nodes, which saves us some heap space.
- Finally, the case class `Configuration` represents concrete assignments of variables to values for expressions.

These ADTs allow expressions to be represented as ASTs. The nodes in these trees represent terms—the main abstraction for programs in program transformation with term rewriting, which we explain in more detail in Sect. 7.6) below. Here, we use AST *node* and *term* synonymously.

Exercise 7.2. Draw an instance of our expression meta-model in Fig. 7.12 for the expression: $\neg((A \wedge \neg B) \vee ((C \wedge \neg D) \vee (\neg C \wedge D)))$. Are different instances possible to represent this expression? In your answer, distinguish between syntactic and semantic equality.

Transformation goal. Our goal is to transform logical expressions into expressions where Identifier nodes whose values are set within a partial assignment (represented by the class `Configuration` in our meta-model) are replaced by the values (i.e., *true* or *false*) of those Identifiers.

```scala
sealed abstract class Expression:
  def & (other: Expression): Expression = other match
    case True => this
    case _ => AND (this, other)

  def | (other: Expression): Expression = other match
    case False => this
    case _ => OR (this, other)

  def unary_! : Expression = NOT (this)

sealed abstract class BinaryExpression (
  val left: Expression,
  val right: Expression
) extends Expression

sealed abstract class UnaryExpression (
  val expr: Expression
) extends Expression

case class NOT (e: Expression) extends UnaryExpression (e):
  override def toString = "!" + e

case class AND (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " & " + r + " )"

case class OR (l: Expression, r: Expression) extends BinaryExpression (l, r):
  override def toString = "( " + l + " | " + r+ " )"

case class Identifier (name: String ) extends Expression:
  override def toString = name

given StringToExpression: Conversion[String, Identifier] with
  def apply (s: String) = Identifier (s)

case object True extends Expression:
  override def & (other: Expression) = other
  override def unary_! = False
  override def toString = "TRUE"

case object False extends Expression:
  override def | (other: Expression) = other
  override def unary_! = True
  override def toString = "FALSE"

case class Configuration (
  val name: String,
  val identifierValues: Map[Identifier,Expression])
```

source: dsldesign.expr.scala/src/main/scala/dsldesign/expr/scala/adt.scala

Figure 7.12: *ADTs for a simple expression language in Scala*

Transformation requirements. We require all Identifiers to be visited and replaced if part of the partial assignment. The rest of the expression instance, especially its structure, should remain unchanged. In principle, the transformation could be an in-place transformation, but we require the

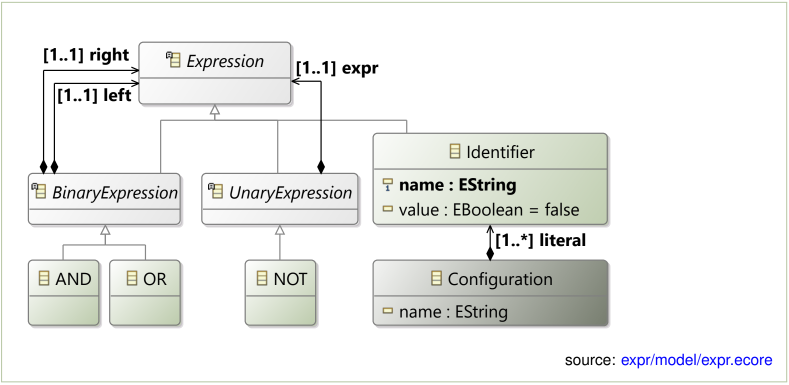


Figure 7.13: Visualization of the expression language in Fig. 7.12

source models to be immutable, so we need a copying transformation. It also needs to be parameterized—the partial assignment is the parameter.

Transformation technology. Since our source and target models, and their meta-model, are defined in Scala, we also implement the transformation in Scala. Using the strategic-programming framework Kiama in this case lets us avoids having to implement recursive methods that traverse the source model (e.g., in a top-down, depth-first-search manner) and copy the visited nodes into nodes organized in the same structure, with the exception of `Identifier` nodes that are part of the partial assignment. Kiama relieves us from having to create such recursive methods, which can become quite difficult when we need multiple traversals or change the traversal based on the sub-tree structure, and so on.

Transformation implementation. Figure 7.14 shows our implementation. We use the trait `CopyingParameterizedTrafo`, which is part of the book-accompanying libraries (`dsldesign.scala.emf`) to simplify using Scala. The trait requires setting the parameter `p: Configuration` upon instantiation. Since this way we have some state in the object (i.e., the configuration), we write the transformation as a Scala class instead of an object. The `run` method calls Kiama's rewriter, which creates the target model by applying `constantPropagationRule` to the elements of the source model to which it is applicable, while copying the rest of the model to leave it unchanged.

Our rule in the function `constantPropagationRule` does pattern matching on terms (nodes in the AST); specifically, it checks for an Identifier node. If its name is in the configuration parameter, then the Identifier is replaced with the respective value, that is, either the expression *true* or *false*.

The rule is applied using Kiama's strategy `everywherebu`. It applies the rule to all (sub-)terms of the root node (subject term) of the source expression (for which we do the constant propagation) in a bottom-up manner.

Exercise 7.3. For our expression language, extend the algebraic data types with a binary expression type IMPLIES. So, logical expressions can be defined using the *implies* ($\rightarrow$) operator (i.e., $A \rightarrow B \equiv \neg A \vee B$). Add a convenience method `implies` also to the class `Expression` and to `True`, which allows an easy instantiation of

```
class ConstantPropagation (val p: Configuration) extends
  CopyingParameterizedTrafo[Expression, Configuration, Expression]:

  val constantPropagationRule = everywherebu {
    rule[Expression] {
      case id@Identifier (_)
        if p.identifierValues.keySet.contains (id) =>
          p.identifierValues.get (id).get
    }
  }

  override def run (self: Expression): Expression =
    Rewriter.rewrite (constantPropagationRule) (self)
```

source: expr.scala/src/main/scala/dsldesign/expr/scala/transforms/ConstantPropagation.scala

Figure 7.14: A constant-propagation transformation in Kiama

implications, similarly to the other methods in the class. Optionally, do the same for Exclusive OR (a.k.a. `XOR` or $\oplus$).

Case Study 2: Simplifying Logical Expressions

Let us now create a transformation to syntactically simplify logical expressions. We reuse our ADTs to represent expressions in an AST. Proper expression simplification is a hard problem, when it should be sound (the semantics remains the same) and complete (no shorter expression with the same semantics is possible). We use a more tractable method that works purely on the expression syntax. It is sound, but not necessarily complete.

Transformation goal. The goal of expression simplification is to preserve the semantics of expressions, but to reduce their structural complexity. In our case, the goal is to decrease the number of nodes in an expression.

Transformation requirements. So, we require that after each rule application, the number of nodes is lower. This can be achieved by only creating rules whose left-hand sides—the patterns over source elements to which the rule can be applied—are larger than their right-hand sides.

Transformation technology. Our form of expression simplification can be nicely expressed in Kiama rules for strategic programming in Scala. So, we reuse our ADTs from the previous case study that hold the expressions.

Transformation implementation. Figure 7.15 shows our implementation. We define rules for common simplifications of Boolean expressions. Recall that these implementations are inspired by our work on analyzing systems software, including the Linux kernel. What we actually did was to observe the expressions and manually find parts that we could simplify, which we then abstracted into individual rules. The rules are put into a pattern-matching statement in Scala, where they are applied to nodes in the AST. Each node represents a term.

The application of this rule (function `simplifyRule`) is controlled by Kiama's strategy `innermost`. It belongs to the so-called fixed-point strategies, which exhaustively apply rules to terms (i.e., expression nodes) until no rule is applicable anymore. `Innermost` applies `simplifyRule` repeatedly to the lowest and leftmost (i.e., the “innermost”) sub-term to which the rule applies—that is, where any of the patterns in the rule matches.

Intuitively, the strategy starts with the subject term (its parameter), then goes down to the lowest ancestor, taking the leftmost child as a route, and tries to apply `simplifyRule`. If not applicable, it goes to the next child until one is applicable, and otherwise goes up. This strategy makes sure that our rules are applied as often as possible, until none of them matches the subject term or any of the ancestor terms.

```
object SimplifyExpression extends CopyingTrafo[Expression, Expression]:

  val simplifyRule = innermost {
    rule[Expression] {
      case NOT (NOT (a))                  => a
      case NOT (True)                     => False
      case NOT (False)                    => True
      case OR (a, b) if a == b            => a
      case OR (True, a)                   => True
      case OR (False, a)                  => a
      case OR (a, True)                   => True
      case OR (a, False)                  => a
      case AND (True, a)                  => a
      case AND (False, a)                 => False
      case AND (a, True)                  => a
      case AND (a, False)                 => False
      case AND (a, b) if a == b           => a
      case AND (AND (a, b), c) if b == c => AND (a, b)
      case AND (a, AND (b, c)) if a == b => AND (b, c)

      case OR (AND (a, b), AND (c, d)) if a == c => AND (a, OR (b, d))
      case OR (AND (a, b), AND (c, d)) if b == d => AND (b, OR (a, c))
      case OR (AND (a, b), AND (c, d)) if b == c => AND (b, OR (a, d))
      case OR (AND (a, b), AND (c, d)) if a == d => AND (a, OR (b, c))

      case AND (OR (a, b), OR (c, d)) if a == c => OR (a, AND (b, d))
      case AND (OR (a, b), OR (c, d)) if b == d => OR (b, AND (a, c))
      case AND (OR (a, b), OR (c, d)) if b == c => OR (b, AND (a, d))
      case AND (OR (a, b), OR (c, d)) if a == d => OR (a, AND (b, c))

      case OR (a, AND(b, c)) if (a==b || a==c) => a
      case OR (AND(a, b), c) if (a==c || b==c) => c

      case OR (AND (a, b), OR (c, d))
        if (a==c || a==d || b==c || b==d) => OR (c, d)
      case OR (OR (c, d), AND (a, b))
        if (a==c || a==d || b==c || b==d) => OR (c, d)

      case OR (NOT (a), b) if a==b => True
      case OR (a, NOT (b)) if a==b => True
    }}

  override def run (self: Expression): Expression =
    Rewriter.rewrite (simplifyRule) (self)
```

source: expr.scala/src/main/scala/dsldesign/expr/scala/transforms/SimplifyExpression.scala

Figure 7.15: Simplifying logical expressions using the `innermost` strategy in Kiama

Exercise 7.4. Extend the simplification rule in Fig. 7.15 to handle logical implications as well. Represent implications with a new case class `IMPLIES`. Also create test cases.

Case Study 3: Transforming Logical Expressions into the Conjunctive Normal Form

A common transformation for Boolean expressions is to convert them into the conjunctive normal form (CNF). CNF refers to the formula being in a special syntactic form. A Boolean formula is in CNF when it is a conjunction of clauses. Clauses are disjunction of literals, and literals are variables, their negation, or just true or false values. Formally, an expression in CNF looks like this: $\bigwedge_i \bigvee_j (\neg) x_{ij}$.

Reasoners, such as Satisfiability (SAT) solvers, usually require expressions to be in CNF. It allows efficient algorithms, a standard format called Dimacs, and quickly rules out satisfiability when already one clause is not satisfiable. There are some SAT solvers that do not require CNF, but the majority require that others have taken care of the conversion already. Importantly, any Boolean formula can be converted into CNF. However, this conversion can be quite complex, especially for disjunctions, where the CNF transformation can virtually explode the expressions in size, requiring further tactics to make the transformation tractable, such as the transformation by Tseitin [68]. However, here we will realize the most simple conversion from Logic textbooks, where one applies certain logical laws exhaustively.

Transformation goal. The goal is to transform small Boolean expressions into CNF. By 'small,' we refer to expressions that can still easily be represented as object structures in the heap memory and do not cause stack overflow errors when traversed using recursive functions (as is very common in functional programming languages), and that do not cause exponential explosion in size and time when converted into CNF. Handling large expressions requires different representations and transformation techniques than those we use in the book.

Transformation requirements. For brevity, the transformation should not introduce auxiliary variables into the formula. The latter is necessary to make the CNF conversion scalable for large formulas, which can easily explode. Here, we avoid such variables and rather want to illustrate a clean transformation that applies the logical laws as transformation rules as directly as possible.

Transformation technology. CNF conversion can largely be expressed in Kiama rules for strategic programming in Scala. So, as previously, we reuse our ADTs for expressions (Fig. 7.12).

Transformation implementation. Figure 7.16 shows our implementation. The most important rule is `distributiveRule`, which applies the distributive laws (e.g., $X \vee (Y \wedge Z) \Leftrightarrow (X \vee Y) \wedge (X \vee Z)$). Applied repeatedly, they push conjunctions up in the expression AST, and disjunctions down.

The other rules should be applied before, including: De Morgan's laws (e.g., $\neg(X \wedge Y) \Leftrightarrow \neg X \vee \neg Y$) in `demorgansRule`, which push the negations down to the Identifiers, so that they only appear as part of a literal; the `doubleNegationRule`, which removes double negations; and the `valueNegationRule`, which resolves negations of `True` and `False`.

```scala
object ExpressionToCNF extends CopyingTrafo[Expression, Expression]:

  val demorgansRule = reduce {
    rule[Expression] {
      case NOT (AND (x, y)) => OR (NOT (x), NOT (y))
      case NOT (OR (x, y)) => AND (NOT (x), NOT (y))
    }
  }

  val doubleNegationRule = reduce {
    rule[Expression] {
      case NOT (NOT (x)) => x
    }
  }

  val valueNegationRule = everywheretd {
    rule[Expression] {
      case NOT (True) => False
      case NOT (False) => True
    }
  }

  val distributiveRule = innermost {
    rule[Expression] {
      case OR (x, AND (y, z)) => AND (OR (x, y), OR (x, z))
      case OR (AND (x, y), z) => AND (OR (x, z), OR (y, z))
    }
  }

  def run (self: Expression): Expression =
    Rewriter.rewrite(
      demorgansRule <*
      doubleNegationRule <*
      valueNegationRule <*
      distributiveRule) (self)
```

source: expr.scala/src/main/scala/dsldesign/expr/scala/transforms/ExpressionToCNF.scala

Figure 7.16: *CNF conversion of logical expressions in Kiama*

We use Kiama's rewriter to apply these rules repeatedly. They are combined with the combinator `<*`, which expresses sequential composition. It creates a strategy that applies the first strategy first, and when it succeeds it applies the second strategy to the rewritten term. It fails otherwise, so the second is only applied when the first succeeds. In our case, the rules all succeed regardless whether they rewrote the subject term or not.

It is important to first exhaustively apply De Morgan's laws, since rewriting can produce double negations, while the latter cannot produce new sub-expressions where De Morgan's laws could be applied. Both rules use Kiama's `reduce`, which is a fixed-point strategy, which exhaustively apply rules to terms (i.e., expression nodes) until no rule is applicable anymore. Reduce works recursively, for each term it applies the rule t "repeatedly to subterms until it fails on all of them" according to Kiama's documentation. Thereafter, we apply the `valueNegationRule`, which only requires one full traversal of the tree, achieved via the strategy `everywheretd`. It tries to apply the rule to every term (i.e., node) in the expression AST.

Finally, we apply the `distributiveRule`, which again requires a fixed-point strategy. We could use `reduce` again, but prefer the strategy `innermost`, since it usually has a performance advantage [62]. The former repeatedly traverses the whole AST from the top, searching for terms to apply the rule. But, as already explained for the previous transformation, `innermost` starts with the 'innermost' term, which is the leftmost leaf node in the AST. It then traverses the tree up until it can apply the rule to reduce the innermost terms first; it only stops when the rule is not applicable anymore anywhere in the tree. So, the subject term (i.e., root node) is transformed last.

A hint on testing. Note that getting a transformation like this one right can be tricky, especially when you want to realize it as concisely as possible as in our implementation. To this end, we created special test cases, relying on generated expressions and a method that checks whether an expression is in CNF. We explain the details in Sect. 7.8 below.

Exercise 7.5. Extend the CNF transformation in Fig. 7.16 to accept and transform expressions with implication operators (`IMPLIES`) as well. Represent implications with a new case class `IMPLIES`, as in the previous exercise. Create test cases.

Exercise 7.6. Extend the CNF transformation so that, after conversion to CNF, it reduces clauses (i.e., disjunctions of literals) that contain a $true$ or $false$ literal. An example would be $(A \vee B) \wedge (B \vee true) \wedge (C \vee false)$, where the second clause can be removed, and the third clause can be reduced to (C). Create test cases.

Exercise 7.7. Investigate the actual performance of the strategy `innermost` compared to `reduce` for the rule `distributiveRule`. While the former usually has a performance advantage, it is actually not clear whether this advantage also applies to our CNF transformation. Provide concrete arguments based on the distributive rule for which strategy will be faster. Write a small performance test that compares the performance of both strategies on generated expressions. You can use our expression generator from Fig. 7.20.

7.6 Transformation Technologies

An incredible number of transformation technologies exists from the software engineering and the programming language communities [39, 21, 70]. We now only focus on M2M transformation technologies from the Modelware (model transformation) and Grammarware (program transformation) fields. We discuss M2T in detail in Chapter 9.

Before looking at the technologies, let us first see what a transformation can look like when it is written in a GPL that does not have any of the abstractions suited for transformations—in plain C. Figure 7.17 shows an excerpt from the Linux kernel configurator, which we will examine in detail later (Sect. 11.2). Like our case study in Sect. 7.5, it is about simplifying logical expressions. Here, the source and target models—the expressions—are defined in C structs instead of meta-models or algebraic data types.

```
 1 /*
 2  * Recursively performs the following simplifications in-place (as well
 3  * as the corresponding simplifications with swapped operands):
 4  *
 5  * expr && n  ->  n
 6  * expr && y  ->  expr
 7  * expr || n  ->  expr
 8  * expr || y  ->  y
 9  *
10  * Returns the optimized expression.
11  */
12 static struct expr *expr_eliminate_yn(struct expr *e){
13    struct expr *tmp;

15    if (e) switch (e->type) {
16    case E_AND:
17       e->left.expr = expr_eliminate_yn(e->left.expr);
18       e->right.expr = expr_eliminate_yn(e->right.expr);
19       if (e->left.expr->type == E_SYMBOL) {
20          if (e->left.expr->left.sym == &symbol_no) {
21             expr_free(e->left.expr);
22             expr_free(e->right.expr);
23             e->type = E_SYMBOL;
24             e->left.sym = &symbol_no;
25             e->right.expr = NULL;
26             return e;
27          } else if (e->left.expr->left.sym == &symbol_yes) {
28             free(e->left.expr);
29             tmp = e->right.expr;
30             *e = *(e->right.expr);
31             free(tmp);
32             return e;
33          }
34       }
35 ...
```
source: https://github.com/torvalds/linux/blob/master/scripts/kconfig/expr.c

Figure 7.17: Expression simplification in C from the Linux kernel

Without a transformation technology, one needs to write recursive functions that dive down the AST of the models, as you can see in the excerpt. An alternative from object-oriented languages is the visitor design pattern, which is not very concise, however (one needs to create several classes). In any case, when using recursion or the visitor pattern, transformations are difficult to understand and write. They are even more difficult to maintain, since they do not separate the traversal and the rewriting of AST nodes.

Also observe how pattern matching on the AST nodes is realized in this excerpt. In our case study in Scala using the Kiama framework, line 17 to 35 would only be two lines. This illustrates the benefit of having pattern-matching support.

Tools, Libraries, and Languages

Addressing these issues, a large number of libraries and languages specialized for manipulating models exist. They mainly differ in the way they provide facilities for traversing models and for rewriting source into target elements. GPLs not only with pattern matching, but also with higher-order functions are also useful—especially when models are represented as algebraic data types. For this reason, transformations of syntax trees are traditionally convenient to write in languages such as Scala, F#, or Haskell, with diverse supporting libraries (internal DSLs) that simplify writing transformations. It helps especially when they take over control of the order in which the rules are executed. Besides that, a number of specialized external DSLs for writing model transformations exist, such as QVT-O, ATL, or Xtend, which we illustrated above in Tbl. 7.1.

Such specialized transformation tools, libraries, and languages are usually associated with certain claims. Among others, they are claimed to: be easy to use, provide usable abstractions to developers, be scalable to large source models, provide traceability, and support many different automation use cases, including incremental updates or even bidirectional transformations.

The implemented transformations are usually claimed to be more comprehensible and easy to evolve, maintain, and reuse—as opposed to being implemented in ordinary GPLs. In our opinion, the available tools, libraries, and languages have only partially delivered on these claims.

Model-Transformation Technologies

Surveys [39, 21] classify M2M model-transformation tools and technologies into the following categories.

- *Imperative* transformation technologies, where the manipulation of model elements is very directly expressed in terms of imperative statements. These transformations are at a rather low level. Therefore, they are also called *direct manipulation* techniques. Just using a GPL to implement the transformation falls under this category, even when enhanced with some libraries (i.e., internal DSLs) for tracing or controlling the rule application. Examples of M2M technologies realized as external DSLs are QVT-O and Xtend.[8]
- *Relational* transformation technologies, where the manipulation of model elements is performed by the engine based on a mapping definition among source and target models. The mapping is explicit in the transformation definition; the actual manipulation of a source model is implicitly defined through it when the engine translates the mapping definition into manipulations. Examples are JTL [16] and QVT-Relations [54]. Less expressive than relational techniques, but following a similar idea, are *structure-driven* techniques, which are most suited for rather simplistic

[8]Xtend is more than a DSL, it is in fact a GPL, but with specific support for transformation. Since it is mainly used for that, it is probably fair to also call it a model-transformation DSL.

1-to-1 mappings between source and target elements, without requiring iterations or even fixpoint calculations.

- *Graph-based* transformation technologies, a.k.a. graph transformations [25], rely on algebraic graph transformation [3], a technique to algorithmically obtain a new graph from an existing graph, which has many applications beyond software engineering. Graph transformations formally describe the modification of graphs, in our case models in abstract syntax. In graph-based transformation technologies, the transformation rules are pairs of graph patterns, often using a graphical syntax.
- *Hybrid* transformation technologies combine aspects from the other three technologies above. Examples are the Atlassian Transformation Language (ATL), GrGen [30], Viatra2 [69], and Henshin [4, 67]. The latter is primarily a graph-based transformation technology, but optionally provides concepts from imperative technologies to allow extensive rule application control. We illustrate Henshin below.

Since the field is huge, we refer to surveys [39, 21] for further details on these categories of M2M transformations and for more technologies. There are also more-specific surveys which, for instance, focus on bidirectional transformation technologies including QVT-Relations or JTL [26, 34, 23].

Example 23. Henshin is a hybrid transformation technology. It offers a graphical and textual syntax to define graph-transformation rules. The rules declare patterns over model elements, specifying which elements to remove, add, replace, and so on in models. It is especially suited for in-place endogenous transformations. Henshin's core supports unidirectional transformations, but an extension for bidirectional transformations exists (HenshinTGG).

The example below is an adaptation of a small part of our transformation from Sect. 7.5 that simplifies logical expressions. Specifically, it represents the rewriting of $((A \wedge B) \wedge C)$ when $B = C$ in line 18 of Fig. 7.15.

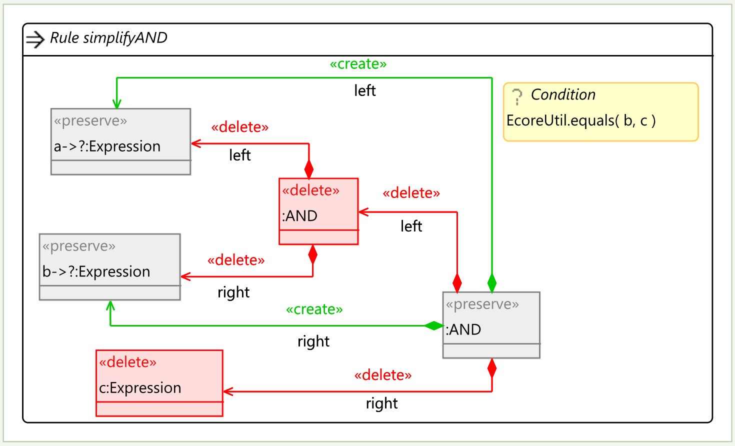


Figure 7.18: A model-transformation rule in Henshin

Program-Transformation Technologies

Program transformations, attributed to the community around *Grammarware*, transform programs instead of models or other data. Since we learned that models and programs are very close, this can hardly be seen as a distinguishing criteria. Instead, we see the following differences.

Differences to Model Transformation The main difference is probably that program transformations are based on mathematics-oriented concepts, such as term rewriting, attribute grammars, and functional programming [9]. In contrast, model transformations usually target object-oriented programming and adopt an object-oriented approach for representing and manipulating models.

Second, model transformations, as they relate models to each other, are often expected to record traceability links, which is not done in program transformation [21]. Traceability links are meta-data about the origins of the generated output elements, used for debugging, understanding, documentation, or auditing purposes.

Third, model-transformation technologies often come as external DSLs, while program-transformation technologies come as internal DSLs (via a library). This can be seen well in our case studies in Sections 7.2 and 7.5.

Fourth, model transformations are programmed against meta-models. These constrain valid instances and usually provide stronger guarantees than are offered by algebraic data types. So, checking for valid instances does not need to be done by model transformations, while a program-transformation developer needs to take more care [43].

Term Rewriting Program transformation commonly relies on term rewriting [9]. Kiama [62], which we used above in Sect. 7.5, is based on the rewriting paradigm. Other examples are Stratego [71] and TXL [19, 20].

In term rewriting you abstractly represent programs as *terms*. Like ASTs, terms are ordered trees. That makes them particularly useful to represent source code, where you have lists of statements, declarations, or expressions. These are the child nodes of a node representing a method or a block in source code. Still, term syntax can sometimes be difficult to comprehend. It is especially challenging when inputs (programs or program fragments) are large [70]. Notably, terms in term rewriting are limited to abstract syntax, which can be complex sometimes.[9]

The other important abstraction is *rewrite rules*. They define source patterns and target patterns. If a rule is applicable, then source and target elements are in a reduction relationship. The source is called the *redex*, the target is called the *reduct*.

The source patterns are used to check applicability. However, pattern matching (as in Scala) is often limited. For instance, one usually cannot deal with associativity or commutativity of term structures. Various extensions

[9] Stratego exceptionally allows rules to be defined over concrete syntax as well [71].

were proposed for this reason, such as associative-commutative matching in Maude, or recursive patterns for matching sub-trees. Furthermore, transformation rules often need additional conditions (in addition to the patterns) to check applicability. This can, for instance, be seen in our expression simplification case study in Sect. 7.5.

Term rewriting is typically context-free. However, when using it in modern programming languages, as we demonstrated with Scala, rules do not need to be context-free, but can access their context (e.g., a partial assignment in our constant-propagation transformation in Sect. 7.5). In fact, term rewriting has long been limited to context-free rules, but this has been eliminated by modern rewriting systems and languages, as in our case. Since we used Scala, the rules are closures and can access context, as shown in our case study on constant propagation for logical expressions in Sect. 7.5).

A rule represents one transformation step. But to transform programs, you usually need many steps, so multiple rules need to be applied multiple times. Like in model-transformation technologies, the control over rule application is often separated from the rules themselves. In the simplest case, rule application can be based on lazy function evaluation, as found in many functional programming languages, such as Scala.

Rule Application Control In general, rule application control is classified into: fixed application order, automated dependency analysis, goal-driven, strategy menu, or programmable [70]. In the latter, called strategic programming, rule application is controlled by a program in a so-called strategy language.

Strategic Programming The paradigm of strategic programming, realized for example by Stratego and Kiama, abstracts programmable rule application control into so-called strategies. According to Bravenboer et al. [9]: "A strategy is a little program that makes a selection from the available rules and defines the order and position in the tree for applying the rules." Many different strategies exist, as well as strategic-programming frameworks. For a good overview we recommend the survey by Visser [70].

7.7 Guidelines for Writing Transformations

Guideline 7.1 *Choose the right technological space and transformation paradigm.* You choose the technological space based on the representation of your source and target models, as well as the features of transformation technology you need—with the help of this chapter, of course. Recall that we showed how transformations can be simply implemented in Java or any other major GPL, such as Scala. Since our meta-models are made with EMF, they get Java model infrastructures for free, and all the persistence code is generated and readily available. We can simply load model data into memory objects and manipulate them using Java. However, some specialized features are useful when writing transformations, especially if you want a concise and declarative realization of rules. Then you choose between model- and program-transformation technologies. In this chapter, we conveyed that program transformation, while limited to endogenous transformations, might be better for models at lower levels of abstraction (i.e., programs) with more complex transformation logic. Especially if you transform models that hold structures similar to expressions, you should use program-transformation technology. Here, it seem more natural to use program transformation with term rewriting. While there are transformations on expressions in the ATL transformation zoo[10] (e.g., the transformation "Truth Tables to Binary Decision Diagrams"), expression transformation can be written much more concisely with pattern matching using ADTs, however.

Guideline 7.2 *Start with examples.* It is always good to start with an example for transformations. For complex targets, you might want to convert your example manually to understand the mechanism. Alternatively, if your transformation is really complex or when you need to work on it collaboratively with someone else, you could write down the transformation in natural language. It helps to break down the transformation logic into smaller rules.

Guideline 7.3 *Use test-driven development.* Even better than just using examples, already express the examples as test cases. Do not underestimate testing for transformations, which is more challenging than for ordinary programs. While the transformation implementation as the unit under test is often simpler than typical units under tests in ordinary software development, the input and output test data are actual models, which poses new problems. There is also little explicit support for testing, so start early with testing, ideally adopting a test-driven development style. We discuss testing shortly in Sect. 7.8.

Guideline 7.4 *Make the types of the source and target explicit via meta-models or ADTs.* The source and target adhere to certain types against which the transformation logic is programmed, so we make the types explicit. When you write a transformation, you often have the types implicitly in mind and put them into the logic. But, here we have models adhering to meta-models (types), which allows us to have a full infrastructure for processing models, and then we can program our transformation. This not only helps when implementing transformations, but we can also check whether transformations are complete (e.g., cover the complete meta-model).

Guideline 7.5 *Document the transformation goal.* In the transformation implementation, the goal is only implicitly given in the rules. Especially for optimization, the goal can be declared.

Guideline 7.6 *Be wary of skewed transformations.* You usually want to separate horizontal transformations and vertical transformations. However, it might be sufficient to separate those parts into separate transformation rules. Then, it might be OK. The transformation would then still be skewed, which usually hinders comprehensibility, so you might want to avoid that. But apply common sense here.

[10] https://www.eclipse.org/atl/atlTransformations, seen 2022/09

Guideline 7.7 *Use immutable data structures.* Or, at least ensure that the source instances cannot be changed accidentally. In Scala we use case classes, which prevents accidental modification of the source instead of the target. Dedicated model-transformation languages such as ATL or QVT-O even disallow modifying the source. The source is read-only, and the target is write-only.

Guideline 7.8 *Emulate exogenous transformations with union languages.* Some transformation languages, chiefly those based on the rewriting paradigm (hereunder TXL and Scala's Kiama) only allow copying, thus endogenous, transformations. In these languages, it is necessary to create a union of the target and source languages, in order to implement exogenous transformations as endogenous transformations within the union language.

Guideline 7.9 *In transformation rules, put computation into the right context.* We have often seen transformations in which computation is put into a wrong or inconvenient context. By 'context' we mean the input pattern, or place in the source model (e.g., a specific meta-class) where a rule is applied. An example would be object instantiation, where putting the object in the right place in the partonomy or establishing links to other objects can be inconvenient (e.g., requiring retrieval of objects from the trace model or manually keeping track of them) or impossible when not done in the right context. Recall from Sect. 5.1 that choosing the context (context class or starting type) wisely is also important when writing constraints.

Guideline 7.10 *Use transformation chaining for separation of concerns.* We mentioned chaining of transformations a couple of times in this chapter. Decomposing a large and complex transformation into smaller ones fosters modularity, reuse, and maintainability. It is common to have some M2M transformations ending with an M2T transformation. The latter can be seen as a view, while the M2M transformations perform the actual transformation of some source model. Since these transformations are changed at different frequencies and often by different developers—views are often changed more frequently, and the changes do not affect how the models are transformed, but rather how they are presented—one should separate them. In particular, there should be no calculations in the M2T transformation. For instance, if you use a template-based M2T, then the template should just present values from the transformed models, not doing any further calculations beyond just very simple ones (e.g., to change a date format).

7.8 Quality Assurance

Regardless of whether you use a dedicated transformation technology or an ordinary programming language, you will need to quality-assure the transformation to prevent faults. Quality assurance can be done using static methods, such as manual code reviews or automated static analysis, or it can be done using dynamic techniques, which mainly means testing. As for static techniques, we believe that code reviews are similarly applicable as for ordinary software projects, while there are almost no automated static analysis techniques. The latter exist for the field of compiler verification [1, 36], so for compilers of GPLs. These are hardly applicable for the kinds of model and program transformations for DSLs we describe in this book. So, your main quality assurance technique for transformations will be testing.

Since the dynamic semantics is a substantial part of your DSL engineering project, you can expect a substantial fraction of resources to be devoted to testing [57]. The fraction is likely even larger than in common software projects. The main reason is that the core parts of the DSL development are supported by very efficient model-driven tools that make you proceed faster. However, very little specific support exists for testing of model or program transformations, so you are down to regular development speed.

However, there is an interesting tradeoff compared to testing of ordinary programs: the input and output test data (i.e., models) are much more complex, while the code is often simple, except for difficult transformations. Our transformation of Boolean expressions into CNF above (Sect. 7.5) can be seen as a more difficult one.

Case Study: Testing our Transformation of Logical Expressions

Let us start with an example where we create an automated white-box test of the most complicated transformation from this chapter: our transformation of logical expressions into the conjunctive normal form (CNF) in Sect. 7.5. We will discuss the random generation of input data (i.e., models representing expressions) and how to write an oracle, which tells us whether the output data is correct.

Trying to cover the different rules, we create simple test data together with expected outputs (test oracle) as shown in Fig. 7.19. We use Scalatest, but the test framework is not really essential; we could have used JUnit as well. The source models for these rules can be simple, and we instantiate them using the simple internal DSL for expressions we defined with algebraic data types (Fig. 7.12), which makes this part and the oracles really concise (just one line of code for each expression).

The last rule is the most complicated, since it relies on fixpoint generation and also needs the previous rules to be applied to an expression for meaningful results. We randomly generate input data using a custom generator. We could have used ScalaCheck as we did in Fig. 6.16 in Sect. 6.7, but we also illustrate how one can write a generator from scratch here. In fact, for such smaller languages like `expr`, we believe that writing a random-instance generator causes less overhead than having to set up and configure an instance generator for meta-models or ADTs.

Figure 7.20 shows our hand-crafted instance generator. What we need to do is to randomly generate identifier names, which the first part of the method `generateExpr()` is about. Thereafter, we randomly create nodes of the tree recursively, where the node type is determined randomly, down to a maximum nesting depth, which is a reliable stopping criterion for expressions. The number of nodes would be more brittle, since one

```scala
class ExpressionToCNFSpec extends
  org.scalatest.freespec.AnyFreeSpec,
  org.scalatest.matchers.should.Matchers:

  val transform = ExpressionToCNF

  "test De Morgan’s rule" in {
    val e = "a" | !("x" & !("y" & "z"))
    val res = rewrite (transform.demorgansRule) (e)
    res should equal ("a" | (!"x" | (!(!"y") & !(!"z"))))
  }

  "test double negation rule" in {
    val e = "a" | !(!"x")
    val res = rewrite (transform.doubleNegationRule) (e)
    res should equal ("a" | "x")
  }

  "test negation of values rule" in {
    val e = "a" & !True | "b" & !False
    val res = rewrite (transform.valueNegationRule) (e)
    res should equal ("a" & False | "b")
  }
  ...
```

source: expr.scala/src/test/scala/dsldesign/expr/scala/transforms/ExpressionToCNFSpec.scala

Figure 7.19: Testing individual rules of our transformation of logical expressions into CNF

can easily generate very complex expressions that would crash the CNF conversion and make the tests flaky.

```scala
def generateExpr (maxNumberOfIdentifiers: Int, maxNestingDepth: Int)
  : Expression =
  val r = Random()
  val identifiers = (26 to (maxNumberOfIdentifiers + 25))
    .map { i =>
      (i % 26 + 65).toChar.toString + (if i/26 == 1 then "" else i/26) }
  subexp (maxNestingDepth, identifiers)

private def subexp (depth: Int, ids:Seq[String]): Expression =
  if depth <= 0
    then Identifier (ids (Random.nextInt (ids.size)))
    else Random.nextInt (4) match
      case 0 => Identifier (ids (Random.nextInt (ids.size)))
      case 1 => NOT (subexp (depth - 1, ids))
      case 2 => AND (subexp (depth - 1, ids), subexp (depth - 1, ids))
      case 3 => OR (subexp (depth - 1, ids), subexp (depth - 1, ids))
```

source: expr.scala/src/main/scala/dsldesign/expr/scala/adt/generators.scala

Figure 7.20: Random-expression generator

Figure 7.21 shows the respective test case that generates 50 expressions using our instance generator and checks whether they are in CNF form using the method `isInCNF()`. See the comment in the source code for the idea behind this recursive method. In a way, this method realizes and checks a constraint, similarly to what we do in Chapter 5.

Overall, we believe that this is the most effective way of testing this transformation. Manually creating a substantial number of input and output

```scala
class ExpressionToCNFSpec extends
  org.scalatest.freespec.AnyFreeSpec,
  org.scalatest.matchers.should.Matchers:

  val transform = ExpressionToCNF
  ...

  "test 50 randomly generated expressions" in {
    for i <- 1 to 50 do
      val e = generators.generateExpr( 26, 8 )
      isInCNF (transform.run (e)) should be (true)
  }

  /**
   * The idea is to check that in each path to a leaf, there’s no conjunction after
   * a disjunction anymore; and no disjunction or conjunction after a negation.
   */
  def isInCNF (e: Expression): Boolean =
    def checkAllowedNodeTypesInSubtree
      (node: Expression, conjAllowed: Boolean): Boolean = node match
        case OR (l, r) =>
          checkAllowedNodeTypesInSubtree (l, false) &&
          checkAllowedNodeTypesInSubtree (r, false)
        case AND (l, r) => conjAllowed &&
          checkAllowedNodeTypesInSubtree (l, true) &&
          checkAllowedNodeTypesInSubtree (r, true)
        case NOT (Identifier (_)) => true
        case NOT (_) => false
        case _ => true

    checkAllowedNodeTypesInSubtree (e, true)
```
source: expr.scala/src/test/scala/dsldesign/expr/scala/transforms/ExpressionToCNFSpec.scala

Figure 7.21: Testing whether randomly generated expressions are actually transformed into CNF

data pairs is difficult, especially since it is not so simple to manually check whether an expression is in CNF. The problem is the representation of expressions; for instance, a clause is a binary tree of OR nodes with literals at the bottom, instead of a flat list of literals. So, printing it out will give you an expression with many parentheses. For not absolutely simple expressions, it quickly becomes difficult to observe that the nesting is correct, for instance, that no AND node is a descendant of an OR node, which is not allowed for CNF. Of course, one could transform expressions into flatter representations of CNF expressions, but this would require an exogenous transformation. which is not possible with Kiama or term rewriting in general, so we would need a different transformation technology. Alternatively, we could implement another transformation that flattens the expressions to conform to a special CNF meta-model or respective algebraic data types, but we believe that implementing the static semantics in the method `isInCNF()` is easier in this case.

Unit Testing of Transformations

The most common paradigm to test transformations is white-box testing, where the team that writes the transformation also validates it. Black-box testing we believe is rare. In black-box testing, those who create the tests are not aware of the actual implementation of the transformation. Black-box testing amounts to finding inputs based on some specification of the transformation, or on otherwise assumed behavior.

You usually do not need to write negative test cases. In other parts of the book we advocated creation of both positive and negative test cases. Positive cases use valid input together with the expected output (i.e., the test oracle), while negative test cases use invalid input and check that the transformation fails as expected. Other authors, such as Lämmel [43] also suggest to write both for testing transformations. However, negative test cases are not necessary when proper static semantics are defined for the meta-models, which is what we argue for in the book. Negative test cases would just check the static semantics, so we do not need them here. So, whenever you believe you should have negative test cases, you should rather define them as static semantics, which makes your DSL more robust, since static semantics always need to hold, not just when executing transformations.

With white-box testing you try to create test data that fulfill coverage criteria. In the context of transformations, the following criteria are useful. Coverage criteria stem from experience, and the idea is always to find test data that might reveal errors.

Coverage criteria. The main coverage criteria should be to cover the goal and requirements. The following coverage criteria also make sense for transformations.

- *Meta-model or ADT coverage*: Each meta-model or ADT element should be instantiated at least once in the input models. This criterion of course can have many concrete realizations, including different combinations of these examples:
  - Class coverage (meta-model) or type coverage (ADT);
  - Relationship coverage (meta-model) or reference attribute coverage (ADT);
  - Class (meta-model) or type (ADT) attribute coverage. Since attributes can have many values, this might require equivalence class identification or boundary value analysis to select one or only a few values.
- *Rule coverage:* Each transformation rule should execute at least once.

Many more coverage criteria are possible, but be careful. After you define criteria, you will need to create instances fulfilling them, which can be laborious [51]. Ratiu, Völter, and Pavletic [57] report that it was most effective to start with several models of substantial size (human made) and to mutate them, instead of starting with very small ones and trying to grow them. To manually grow models of interesting size and complexity may prove difficult (as in our logical expression to CNF example above).

Automated random testing. Random testing, also known as fuzzing, relies on randomly generating input models for transformations. We already discussed instance generation in Sect. 5.5, which was done there using ScalaCheck, but other frameworks also exist. Instance generation is a bit easier for trees (as we showed for our `expr` language), but more difficult for complex meta-models that are rather graph-like. Especially generating type-correct instances is difficult, which might require a powerful solver, such as the Alloy Analyzer [2].

Many papers about instance generation for meta-models [10] exist. We refer to Ab. Rahim and Whittle [1], who describe various ones. But again, you might be better off writing your own random-instance generator as we showed above (Fig. 7.20), or using ScalaCheck.

The fuzzing community provides support for randomly creating test input data based on grammars [73]. However, grammars are not as expressive as meta-models or ADTs; for instance, they lack type safety. One can use such techniques to achieve valid models if the static semantics of models are expressible in grammars.

The oracle problem. Test oracles define whether a program output or behavior is correct or not for a specific input. For transformations, creating test oracles is more challenging than for ordinary programs. In addition, showing semantic equivalence between test output and expected output (oracle) for specific test input data can be challenging, especially when the transformation is non-deterministic, so can create slightly different syntaxes for semantically equivalent target models. An example would be introducing auxiliary variables for expression scalability reasons in our CNF transformation, which can have some randomness.

In general, model comparison can be reduced to the graph isomorphism problem. So, a naive implementation would be NP-complete [51]. However, efficient implementations exist that reduce complexity by exploiting additional information, such as identifiers of classes. EMFCompare [11] offers model comparison for Ecore-based meta-models, which you could use.

More pragmatically, it is often sufficient to check for the existence of certain patterns in the output model. These patterns you derive from your transformation implementation. In some transformation languages, especially graph-transformation languages such as Henshin (cf. Fig. 7.18), you provide output patterns, which you can search in the target model. Then, instead of accomplishing model comparison, you use a more lightweight strategy here by checking for patterns. This strategy is often sufficient, since the output meta-model and the static semantics will make sure that your output model is syntactically valid.

Another way to create a test oracle is property-based testing. You test certain properties of the transformation, such as invariants. Again, since we use meta-models or ADTs, the typical type-correctness is already assured. For structurally similar source and target meta-models, an invariant could be correspondence correctness: the presence of a target element for each source element. When you have an inverse transformation in addition to your transformation, or the transformation is even bidirectional, an invariant would be whether you obtain the original source model after transforming in both directions. If the transformation originates from a re-engineering project, you can check target models against an existing legacy transformation—a strategy generally known as regression testing.

Finally, another property of model transformations is robustness. Here, the oracle should merely check whether the transformation crashes, does not finish (e.g., ending up in an endless loop), or exceeds memory constraints. The strategy you can apply is to use the methods above, but just create larger models. For instance, in our instance generator for our `expr` language (Fig. 7.20), you increase the maximum depth of the expression AST via the respective parameter.

Exercise 7.8. What is a suitable test strategy for the transformation in Fig. 7.5 from Sect. 7.2? Use the different aspects we described in this section, such as instance generation, coverage criteria, and test oracle, to come up with a strategy. Describe it in your own words, including exemplary test cases and anything else that you consider relevant.

7.9 A Critical Discussion

Writing model and program transformations will for a long time in the future be a common software engineering activity, given the many use cases for them (Sect. 7.3). However, adopting specialized transformation technology to realize transformations instead of using a mainstream language is not an easy decision. In this light, let us now finally discuss the benefits and the challenges of model-transformation technologies, and have a critical discussion on their future.

Benefits of transformation technologies. Proponents of dedicated model- or program-transformation technologies typically emphasize the following benefits. When engineering transformations, organizations will have lower development and maintenance costs, increased productivity, and a better quality of the implemented transformation. We believe the most relevant benefit is increased maintainability, since:

- Transformation technology fosters a core software engineering principle: separation of concerns. It especially separates the domain from the technology using off-the-shelf DSLs or custom DSLs.
- Models that are transformed are abstractions over complex systems. Given this abstraction, realizing model transformations is easier. This advantage holds for model transformations, not necessarily for program transformations, which are usually at lower levels of abstraction.
- Finally, in most transformation technologies, the rules are declarative, and the source models are immutable, which already prevents many errors.

Challenges of model-transformation technologies. Dedicated model-transformation technologies are also challenging, and the benefits might not be achieved. The additional benefits of dedicated technologies, such as parallelism or traceability, are often not needed in practice.

Consider maintainability, which is challenged by the fact that developers will need to learn a new and specific technology. One of our industrial partners in a company developing MDSE technology said that they implement transformations in Java, since otherwise no one can maintain the transformation anymore when the original developer leaves. It is in fact difficult to hire developers who have the respective knowledge or can quickly learn it. A company developing a commercial language workbench, where we asked for example transformations for teaching and research, said they have only one, which is used for model evolution (when a language evolves, models need to be updated)—which the company believes to be one of the main use cases for model transformations.

On the other hand, there are documented adoptions of dedicated model-transformation technology in industry. The Dutch company ASML maintains 100 model transformations for 22 DSLs [47]. The company FEI Company in the US uses model transformations for their microscope calibration software [14]. The multi-national technology consulting company Altran also reported using model-transformation technology [49]. General Motors in the US is know to have created its own model-transformation engine called DSLTrans [27].

Empirical studies and expert opinions. The community has started discussing the future of model-transformation technology. For instance, Burgueño, Cabot, and Gérard [13] report on an open discussion within the MDSE community, which recognizes the lack of empirical evidence that specialized transformation languages (DSLs, such as QVT-O or ATL) are better than GPLs in practice. A survey that was responded to mainly by academics (i.e., 35 % were solely industrial participants, the rest had an academic affiliation or both) shows that among the respondents, only a very small percentage use a GPL, while most use a dedicated transformation language, with ATL being the most frequently mentioned one, followed by QVT-O. This confirms the interest in such languages by academics, which also explains the large number of technologies that have been developed. The reasons for the low adoption in industry discussed in an open discussion with experts were the lack of professional tool support, which some explained by the lack of customer interest, even when a company wanted to develop one. Another reason was that code generation is becoming less important, but rather the management of models is what industry should focus on. The lack of empirical evidence of the benefit of all the additional features was also pointed out, confirmed by experiences of a discussant stating that creating a medium or large transformation is already cumbersome with the dedicated techniques.

Given the unclear benefit of dedicated transformation technologies, we conducted a controlled experiment comparing ATL, QVT-O, and Xtend [33] ourselves. It was motivated by a previous collaboration with industry, where a student developer re-engineered existing model transformations written in GPLs. The transformations written in the languages ATL and ETL were up to 48 % more concise than those written in Java, but that is within the variance between programmers, Java is known as being rather verbose, and the benefit was almost consumed by custom Java startup boilerplate code. We also performed an evolution step, but the sizes of the modifications were independent of the language, since they did not affect the boilerplate code, which made the transformations in dedicated languages verbose. Given this motivation, we conducted the experiment with student developers, who needed to comprehend, change, and implement (from scratch) model transformation in ATL, QVT-O, and Xtend—the latter was our GPL for reference. Like in the industrial collaboration, there were bigger differences in transformation size between the developers than between the languages. We did not observe a statistically significant benefit among the languages in terms of correctness. Still, a limitation is that the tasks were done on paper, not in an IDE, and we could still identify several aspects where the dedicated transformation languages ATL and QVT-O support the developers, especially with copying objects, identifying contexts, and scheduling computations based on types.

Given these empirical results and opinions, we believe that research into concepts (e.g., meta-models of source and target models, transformation rules, immutability, pattern matching, implicit class instantiation) of model- and program-transformation techniques is valuable, and that we will continue to see concepts and practices in mainstream technology. The future of dedicated transformation languages is less clear to us. This chapter was written based on these insights, showing how to effectively write transformations in mainstream technology, and also showing what transformations can look like when there is no good support (cf. Fig. 7.17).

Further Reading

Classifications of model-transformation technologies. Consult Czarnecki and Helsen [21] for a more systematic study of M2M transformation paradigms. The paper provides a useful, if dated, state of the union regarding model transformation. Most of the technologies mentioned are still available, and the catalog of main characteristics of these technologies has not changed much since then. More recent follow-up work is done by Kahani et al. [39].

Benefits, challenges, and the future. Götz, Tichy, and Groner [31] survey benefits and challenges that are claimed in the literature for dedicated model-transformation technologies. They discuss them in a much more systematic and detailed way than we do in Sect. 7.9. Bucchiarone et al. [12] describe the grand challenges in MDSE, including those related to model-transformation technology.

Formal semantics of languages. Mosses [50] provides a good overview on ways to specify the formal semantics of programming languages. He defines many terms we also use (and define), but mainly from the programming language perspective. Many of the descriptions can easily be adapted to models. But, for instance, when he states that static semantics model compile-time checks and dynamic semantics model runtime behavior, you will intuitively understand the terms, but notice that there is no clear notion of compile time and runtime in MDSE with DSLs. This can be explained, since there is nothing like the standardized compiler architectures from the programming language field in MDSE.

Technological spaces. We discussed the notion of technological space in this chapter. We could already have done that in earlier chapters, but transformations often bridge the spaces, while transformation technology is often specific to a particular space. To illustrate and compare technological spaces, Schauss et al. [59] implement a finite-state-machine DSL, similar to our `fsm` language, in ANTLR, Eclipse EMF, Haskell, Racket, MPS, Rascal, Scala, Sirius, Spoofax, and Xtext.

Specific transformation technologies. Teaching materials and textbooks for specific model-transformation technologies are rare. An exception is QVT-O, where the book by Gronback [32] introduces QVT-O in depth, systematically with examples.

Visser [70, S.6.3] give a good overview of strategic programming. Specifically for the strategic programming framework Kiama, there are many more publications. See the project page[11] for all references, including further discussions on Kiama's term rewriting capability [64, 66] and on its attribute grammar support [64, 63], as well as a description of implementing a compiler in Kiama [65]. Notably, attribute grammars allow using Kiama also for realizing the static semantics of DSLs.

Quality assurance. The survey of model-transformation tools by Kahani et al. [39] details validation support, mainly for testing or simulation. They also provide further references on verification and validation of model transformations.

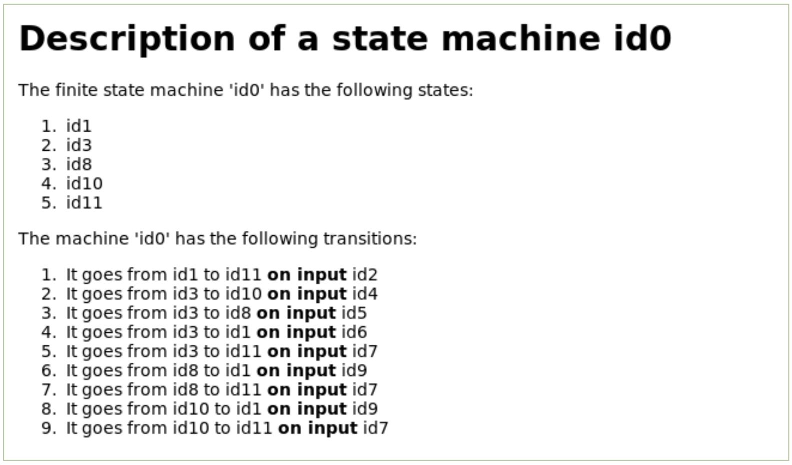
Additional Exercises

The first two exercises that follow demonstrate the chaining of transformations, where you add an M2M transformation before a final M2T transformation. For the latter we refer to examples in Chapter 9.

Exercise 7.9. Implement an M2M transformation that takes an `fsm` model and normalizes all names in the model to be legal Java identifiers. Such a transformation would be a useful addition for our code generator in Sect. 9.4 (see Figures 9.8 and 9.9), which produces invalid code if names of elements in the meta-model contain spaces, illegal symbols, and so on.

Exercise 7.10. In Exercise 9.21 you will build a simple report generator that creates HTML output from `fsm` models. An example output can be found in Fig. 9.18 on p. 354. Let us hypothetically assume that for that generator it would be useful to produce a variant of the output where all the names are *obfuscated*, like in Fig. 7.22. We want to do this without changing the HTML generator but by running an obfuscator on the entire model, and then reusing the generator from Exercise 9.21. Each identifier is replaced by a meaningless name in the format "idN," like in Fig. 7.22. An obfuscator is an in-place model-to-model transformation. In the solution, it might be convenient to implement a function `String obfuscate(String)` that given a string translates it to an obfuscated version in a deterministic way. If you call it twice on the same string of characters

[11] https://inkytonik.github.io/kiama/Research



Description of a state machine id0

The finite state machine 'id0' has the following states:

1. id1
2. id3
3. id8
4. id10
5. id11

The machine 'id0' has the following transitions:

1. It goes from id1 to id11 **on input** id2
2. It goes from id3 to id10 **on input** id4
3. It goes from id3 to id8 **on input** id5
4. It goes from id3 to id1 **on input** id6
5. It goes from id3 to id11 **on input** id7
6. It goes from id8 to id1 **on input** id9
7. It goes from id8 to id11 **on input** id7
8. It goes from id10 to id1 **on input** id9
9. It goes from id10 to id11 **on input** id7

Figure 7.22: An example HTML report generated from an `fsm` instance

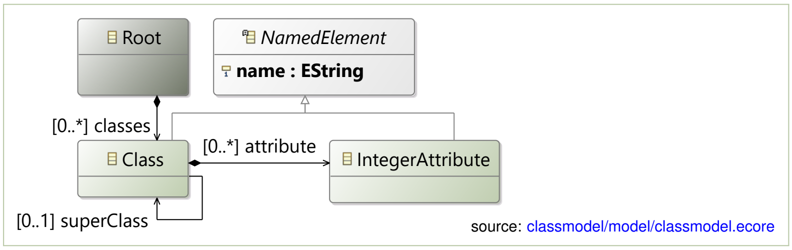


Figure 7.23: A simple class diagrams meta-model, `classmodel`, with class generalization and integer attributes, but no associations

it will return the same obfuscated name, for instance `"id0"`. It guarantees to return a different obfuscated name for different argument strings.

Exercise 7.11. Write an in-place model transformation that adds a new row to an instance of the `Triangle` meta-model presented in Fig. 3.18 from the additional exercises in Chapter 3. The transformation function should take as the first parameter the leftmost entry in the deepest row so far, and as the second parameter the root of the `Triangle` object.

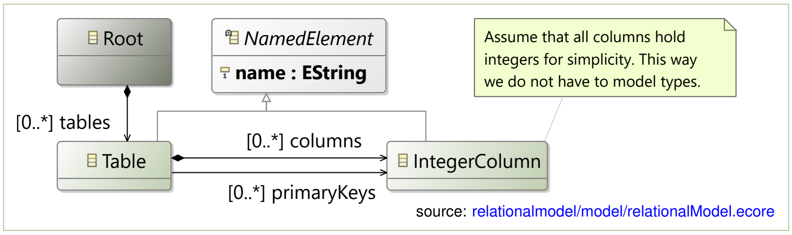

Root
NamedElement
name : EString
Assume that all columns hold integers for simplicity. This way we do not have to model types.
[0..*] tables
Table
[0..*] columns
IntegerColumn
[0..*] primaryKeys
source: relationalmodel/model/relationalModel.ecore

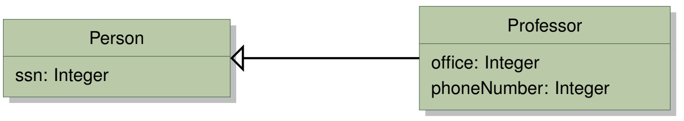
Figure 7.24: A simple meta-model for relational schemas with tables and integer columns, primary keys, but no foreign keys

The following exercises are about transformations and not about DSLs. Notice that structured data transformation is a field that has some application in implementation of languages, but also in other areas.

```
object ClassModelToRelationalModel extends CopyingTrafo[classmodel.Root, RN.Root]:

  val rFactory = RN.RelationalmodelfkFactory.eINSTANCE

  def convertAttribute (self: classmodel.IntegerAttribute): RN.IntegerColumn =
    rFactory.createIntegerColumn before {
      _.setName (self.getName)
    }

  def convertClass (self: classmodel.Class): RN.Table =
    rFactory.createTable before { t =>
      t.setName (self.getName)

      val key = rFactory.createIntegerColumn before { _.setName ("id") }
      t.getPrimaryKeys.add (key)
      t.getColumns.add (key)

      t.getColumns.addAll (self.getAttribute.asScala.map (convertAttribute).asJava)
    }

  override def run (self: classmodel.Root): RN.Root =
    rFactory.createRoot before {
      _.getTables.addAll (self.getClasses.asScala.map (convertClass).asJava)
    }
```

source: relationalmodel.scala/src/main/scala/dsldesign/relationalmodelfk/scala/transforms/ClassModelToRelationalModel.scala

Figure 7.25: A transformation converting from class models to relational models



source: classmodel/test-files/person-professor.xmi

Figure 7.26: An example input class diagram in concrete syntax

Exercise 7.12. Consider the meta-model for extremely simple class diagrams presented in Fig. 7.23. We transform instances of this meta-model to something that can be stored in a (very simplified) relational database. Figure 7.24 presents the meta-model for the database. The implementation of this transformation in Scala is shown in Fig. 7.25.

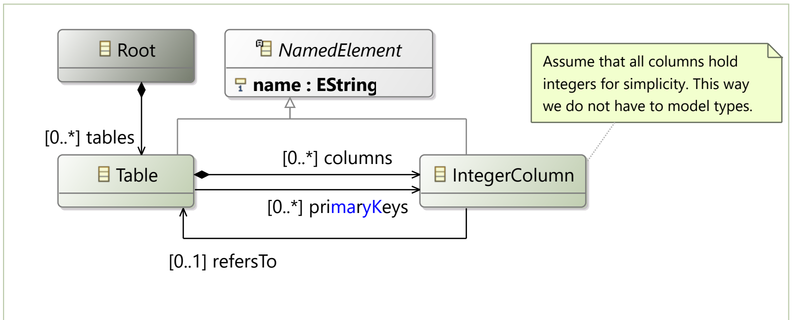
What is the output of running the transformation on the class diagram in Fig. 7.26? Draw the answer in the *abstract syntax* of the RelationalModel.ecore language.

Exercise 7.13. For the transformation shown above, describe in English how to change it along with the associated meta-models to accommodate both integer attributes and string attributes (invalidating the notes in the bottom right corners of the two diagrams).

Exercise 7.14. The implementation of the transformation shown above does not terminate for some inputs. For what inputs will it loop forever or crash? Write (in English) the constraint that rules out such models.

Exercise 7.15. We again transform instances of the meta-model of Fig. 7.23 to something that can be stored in a (very simplified) relational database. This time, we are using the meta-model for the database extended with foreign keys, shown in Fig. 7.27. You may recall we already presented this meta-model in Fig. 5.43 on p. 197. The transformation creates a table for each class. The table contains a primary-key column, always called `"id"`. This key is also included in the primaryKeys. A column is created for every attribute of the class (with the same name as the attribute). Finally, if the class has a super-type, the corresponding table has a column, called `"fk_id"`, to store a foreign key referencing to the table of the super class. The refersTo property points to the table representing the super-class.

In Fig. 7.28 you can find an incomplete implementation of this transformation. Explain what is missing in the blank and complete it. Perform this exercise completely on paper.



source: relationalmodel/model/relationalmodelfk.ecore

Figure 7.27: A simple meta-model for relational schemas with tables and integer columns, primary keys, and foreign keys ('refersTo')

Exercise 7.16. Explain in English what are the main necessary changes to the meta-models and transformations of Figures 7.23 to 7.25 to support binary associations between classes. For clarity of the presentation, it might be useful to sketch the modified parts of the meta-models.

Exercise 7.17. For each of the following kinds of transformation give an example of a practical use case, where it would be applicable, and explain what properties of a programming language you would find useful to implement it (you are allowed to consider any programming languages that you know).

a) A model-to-text transformation

b) A model-to-model transformation

c) An endogenous transformation (an endo-transformation)

d) A text-to-text transformation

References

[1] Lukman Ab. Rahim and Jon Whittle. "A survey of approaches for verifying model transformations". In: *Softw. Syst. Model.* 14.2 (May 2015), pp. 1003–1028 (cit. pp. 275, 279).

[2] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. "Analysis of model transformations via Alloy". In: *Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa. 2007 (cit. p. 279).

```scala
object ClassModelToRelationalModelWithFKs extends CopyingTrafo[classmodel.Root, RN.Root]:

  val rFactory = RN.RelationalmodelfkFactory.eINSTANCE

  def convertAttribute (self: classmodel.IntegerAttribute): RN.IntegerColumn =
    rFactory.createIntegerColumn before {
      _.setName (self.getName)
    }

  def convertClass
     (transformedClasses: mutable.Map[classmodel.Class, RN.Table])
    (self: classmodel.Class): RN.Table =
    transformedClasses.get (self) match
      case Some (table) => table
      case None =>
        rFactory.createTable before { t =>
          t.setName (self.getName)

          val key = rFactory.createIntegerColumn before { _.setName ("id") }
          t.getPrimaryKeys.add (key)
          t.getColumns.add (key)

          t.getColumns.addAll (self.getAttribute.asScala.map (convertAttribute).asJava)

          if (self.getSuperClass != null) {
            /* ----------------------------------------------------------------------------
            |                                                                              |
            |                                                                              |
            |                                                                              |
            |                                                                              |
            ------------------------------------------------------------------------------*/
          }
          transformedClasses.addOne (self, t)
        }

  override def run (self: classmodel.Root): RN.Root =
    val transformedClasses = mutable.Map[classmodel.Class,RN.Table] ()
    rFactory.createRoot before {
      _.getTables.addAll (self.getClasses.asScala.map (convertClass (transformedClasses)).asJava)
    }
```

Figure 7.28: An incomplete M2M transformation in Scala.

[3] Marc Andries et al. “Graph transformation for specification and programming”. In: *Science of Computer Programming* 34.1 (1999), pp. 1–54 (cit. p. 270).

[4] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. “Henshin: Advanced concepts and tools for in-place EMF model transformations”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, pp. 121–135 (cit. p. 270).

[5] Thorsten Berger and Steven She. *Formal Semantics of the CDL Language*. Tech. note. 2010. URL: https://arxiv.org/abs/2209.11633 (cit. p. 250).

[6] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wąsowski. "Feature-to-code mapping in two large product lines". In: *SPLC*. 2010 (cit. p. 259).

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. "A study of variability models and languages in the systems software domain". In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640 (cit. p. 259).

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012 (cit. pp. 251, 256).

[9] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. "Program transformation with scoped dynamic rewrite rules". In: *Fundamenta Informaticae* 69.1-2 (2006), pp. 123–178 (cit. pp. 253, 271, 272).

[10] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. "Metamodel-based test generation for model transformations: An algorithm and a tool". In: *2006 17th International Symposium on Software Reliability Engineering*. IEEE. 2006, pp. 85–94 (cit. p. 279).

[11] Cédric Brun and Alfonso Pierantonio. "Model differences in the Eclipse Modeling Framework". In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34 (cit. p. 279).

[12] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. "Grand challenges in model-driven engineering: An analysis of the state of the research". In: *Software and Systems Modeling* 19.1 (2020), pp. 5–13 (cit. p. 282).

[13] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. "The future of model transformation languages: An open community". In: *Journal of Object Technology* 18.3 (2019) (cit. p. 281).

[14] Zijun Chen. "Evaluation of model transformation testing in practice". MA thesis. TU Eindhoven, 2020 (cit. p. 281).

[15] Noam Chomsky. *Syntactic Structures*. Mouton & Co., 1957 (cit. p. 249).

[16] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. "JTL: A bidirectional and change propagating transformation language". In: *International Conference on Software Language Engineering*. Springer. 2010, pp. 183–202 (cit. p. 269).

[17] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge Into Tools*. CRC Press, 2016 (cit. p. 256).

[18] Olivier Corby and Catherine Faron Zucker. "STTL: A SPARQL-based transformation language for RDF". In: *11th International Conference on Web Information Systems and Technologies*. 2015 (cit. p. 236).

[19] James R. Cordy. "The TXL source transformation language". In: *Sci. Comput. Program.* 61.3 (2006), pp. 190–210 (cit. p. 271).

[20] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. "Source transformation in software engineering using the TXL transformation system". In: *Information and Software Technology* 44.13 (2002), pp. 827–837 (cit. p. 271).

[21] K. Czarnecki and S. Helsen. "Feature-based survey of model transformation approaches". In: *IBM Syst. J.* 45.3 (2006), pp. 621–645 (cit. pp. 234, 251, 256, 267, 269–271, 282).

[22] Susan B. Davidson and Anthony S. Kosky. “WOL: A language for database transformations and constraints”. In: *Proceedings 13th International Conference on Data Engineering*. IEEE. 1997, pp. 55–65 (cit. p. 236).

[23] Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. “A three-dimensional taxonomy for bidirectional model synchronization”. In: *Journal of Systems and Software* 111 (2016), pp. 298–322 (cit. p. 270).

[24] Dragan Djurić, Dragan Gašević, and Vladan Devedžić. “The tao of modeling spaces”. In: *Journal of Object Technology* 5.8 (2006), pp. 125–147 (cit. p. 235).

[25] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific, 1999 (cit. p. 270).

[26] Romina Eramo, Romeo Marinelli, and Alfonso Pierantonio. “Towards a taxonomy for bidirectional transformation.” In: *SATToSE* 1354 (2014), pp. 122–131 (cit. p. 270).

[27] Michalis Famelis et al. “Migrating automotive product lines: A case study”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2015, pp. 82–97 (cit. p. 281).

[28] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. “ConfigFix: Interactive configuration conflict resolution for the Linux kernel”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021 (cit. p. 259).

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995 (cit. p. 251).

[30] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. “GrGen: A fast SPO-based graph rewriting tool”. In: *International Conference on Graph Transformation*. Springer. 2006, pp. 383–397 (cit. p. 270).

[31] Stefan Götz, Matthias Tichy, and Raffaela Groner. “Claimed advantages and disadvantages of (dedicated) model transformation languages: A systematic literature review”. In: *Software and Systems Modeling* 20.2 (2021), pp. 469–503 (cit. p. 282).

[32] Richard C Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009 (cit. pp. 245, 252, 253, 283).

[33] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wąsowski. “Model transformation languages under a magnifying glass – a controlled experiment with Xtend, ATL, and QVT”. In: *26th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2018 (cit. p. 281).

[34] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. “Feature-based classification of bidirectional transformation approaches”. In: *Software & Systems Modeling* 15.3 (2016), pp. 907–928 (cit. p. 270).

[35] Georg Hinkel. “An approach to maintainable model transformations with an internal DSL”. MA thesis. Department of Informatics, Karlsruhe Institute of Technology, 2013 (cit. p. 237).

[36] Tony Hoare. “The verifying compiler: A grand challenge for computing research”. In: *International Conference on Compiler Construction*. Springer. 2003, pp. 262–272 (cit. p. 275).

[37] Arnaud Hubaux, Mathieu Acher, Thein Than Tun, Patrick Heymans, Philippe Collet, and Philippe Lahire. “Separating concerns in feature models: Retrospective and support for multi-views”. In: *Domain Engineering*. Springer, 2013, pp. 3–28 (cit. p. 252).

[38] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993 (cit. p. 253).

[39] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. “Survey and classification of model transformation tools”. In: *Software & Systems Modeling* 18.4 (2019), pp. 2361–2397 (cit. pp. 234, 256, 267, 269, 270, 282, 283).

[40] Paul Klint, Ralf Lämmel, and Chris Verhoef. “Toward an engineering discipline for grammarware”. In: *ACM Trans. on Soft. Eng. and Method. (TOSEM)* 14.3 (2005), pp. 331–380 (cit. p. 236).

[41] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. “Technological spaces: An initial appraisal”. In: *CoopIS, DOA* 2002 (2002) (cit. p. 235).

[42] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. “Reality check for model transformation reuse: The ATL Transformation Zoo case study”. In: *AMT@MoDELS*. 2013 (cit. p. 249).

[43] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018 (cit. pp. 236, 271, 278).

[44] Ralf Lämmel, Eelco Visser, and Joost Visser. “Strategic programming meets adaptive programming”. In: *AOSD*. 2003 (cit. p. 259).

[45] Ralf Lämmel and Joost Visser. “Design patterns for functional strategic programming”. In: *RULE*. 2002 (cit. p. 259).

[46] Robert Lindohf, Jacob Krueger, Erik Herzog, and Thorsten Berger. “Software product-line evaluation in the large”. In: *Empirical Software Engineering* 26.30 (2 2021) (cit. p. 252).

[47] Josh G.M. Mengerink, Alexander Serebrenik, Ramon R.H. Schiffelers, and Mark G.J. van den Brand. “Automated analyses of model-driven artifacts: Obtaining insights into industrial application of MDE”. In: *IWSM Mensura*. 2017 (cit. p. 281).

[48] Tom Mens and Pieter Van Gorp. “A taxonomy of model transformation”. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142 (cit. pp. 235, 256).

[49] A.J. Mooij, M.M. Joy, G. Eggen, P. Janson, and A Rădulescu. “Industrial software rejuvenation using open-source parsers”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2016 (cit. p. 281).

[50] Peter D. Mosses. “Formal semantics of programming languages. An overview.” In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006), pp. 41–73 (cit. pp. 250, 282).

[51] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. “Model transformation testing: Oracle issue”. In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE. 2008, pp. 105–112 (cit. pp. 278, 279).

[52] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. "Where do configuration constraints stem from? An extraction approach and an empirical study". In: *IEEE Transactions on Software Engineering* 41.8 (2015), pp. 820–841 (cit. p. 259).

[53] Siegfried Nolte. *QVT - Operational Mappings: Modellierung mit der Query VSiews Transformation*. Springer, 2010 (cit. p. 245).

[54] Object Management Group. *Query/Views/Transformation Language (QVT)*. https://www.omg.org/spec/QVT. 2016 (cit. p. 269).

[55] Karina Olmos and Eelco Visser. "Strategies for source-to-source constant propagation". In: *Electronic Notes in Theoretical Computer Science* 70.6 (2002), pp. 156–175 (cit. p. 253).

[56] Gergely Pintér and István Majzik. "Program code generation based on UML statechart models". In: *Periodica Polytechnica Electrical Engineering (Archives)* 47.3-4 (2003), pp. 187–204 (cit. p. 251).

[57] Daniel Ratiu, Markus Völter, and Domenik Pavletic. "Automated testing of DSL implementations—experiences from building mbeddr". In: *Software Quality Journal* 26.4 (2018), pp. 1483–1518 (cit. pp. 275, 278).

[58] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer, 2012 (cit. p. 237).

[59] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. "A chrestomathy of DSL implementations". In: *10th International Conference on Software Language Engineering (SLE)*. 2017 (cit. p. 283).

[60] Gehan M.K. Selim, James R. Cordy, and Juergen Dingel. "How is ATL really used? Language feature use in the ATL zoo". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017 (cit. p. 249).

[61] Steven She and Thorsten Berger. *Formal Semantics of the Kconfig Language*. Tech. note. 2010. URL: https://arxiv.org/abs/2209.04916 (cit. p. 250).

[62] Anthony M. Sloane. "Lightweight language processing in Kiama". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2009, pp. 408–425 (cit. pp. 259, 267, 271).

[63] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. "A pure embedding of attribute grammars". In: *Science of Computer Programming* 78.10 (2013), pp. 1752–1769 (cit. p. 283).

[64] Anthony M. Sloane and Matthew Roberts. "Domain-specific program profiling and its application to attribute grammars and term rewriting". In: *Science of Computer Programming* 96 (2014), pp. 488–510 (cit. p. 283).

[65] Anthony M. Sloane and Matthew Roberts. "Oberon-0 in Kiama". In: *Science of Computer Programming* 114 (2015), pp. 20–32 (cit. p. 283).

[66] Anthony M. Sloane, Matthew Roberts, and Leonard G.C. Hamey. "Respect your parents: How attribution and rewriting can get along". In: *International Conference on Software Language Engineering*. Springer. 2014, pp. 191–210 (cit. p. 283).

[67] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. "Henshin: A usability-focused framework for EMF model transformation development". In: *International Conference on Graph Transformation*. Springer. 2017, pp. 196–208 (cit. p. 270).

[68] Grigori S. Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of Reasoning*. Springer, 1983 (cit. p. 265).

[69] Dániel Varró and András Balogh. “The model transformation language of the VIATRA2 framework”. In: *Science of Computer Programming* 68.3 (2007), pp. 214–234 (cit. p. 270).

[70] Eelco Visser. “A survey of strategies in rule-based program transformation systems”. In: *Journal of Symbolic Computation* 40.1 (2005), pp. 831–873 (cit. pp. 234, 255, 257, 267, 271, 272, 283).

[71] Eelco Visser. “Program transformation with Stratego/XT”. In: *Domain-Specific Program Generation*. Springer, 2004, pp. 216–238 (cit. p. 271).

[72] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993 (cit. p. 250).

[73] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. 2022. URL: https://www.fuzzingbook.org (cit. p. 279).

A finished translation is like a tangible,
measurable piece of proof that one has
understood the original perfectly.
Stanisław Barańczak [2]

8 Interpretation

Code generators (Chapter 9) and interpreters are the primary ways to give DSLs a dynamic semantics, to breath meaning into syntax. DSL interpreters are tools that translate the input language piece-by-piece on the fly, like a human simultaneous translator from Danish to German during an interview or a press conference. As a result the input model "executes." Code generators translate the input model entirely in one go into some target language, like a human translator of books from Danish to German. The resulting code can be compiled and executed by the target language's tools. Both interpreters and code generators are special kinds of transformations (Chapter 7).

The overall design of a code generator is largely inspired by compilers. Figure 8.1 summarizes this architecture as a pipeline, from the top left: syntax processing, static semantics processing, translation into intermediate representations and eventually outputting binary code or bytecode. Interpreters typically follow the same architecture, until the translation phase, when the representation is executed instead of being translated. The languages using virtual machines combine both ideas: first compile to an intermediate bytecode representation, then use a byte code interpreter to execute it.

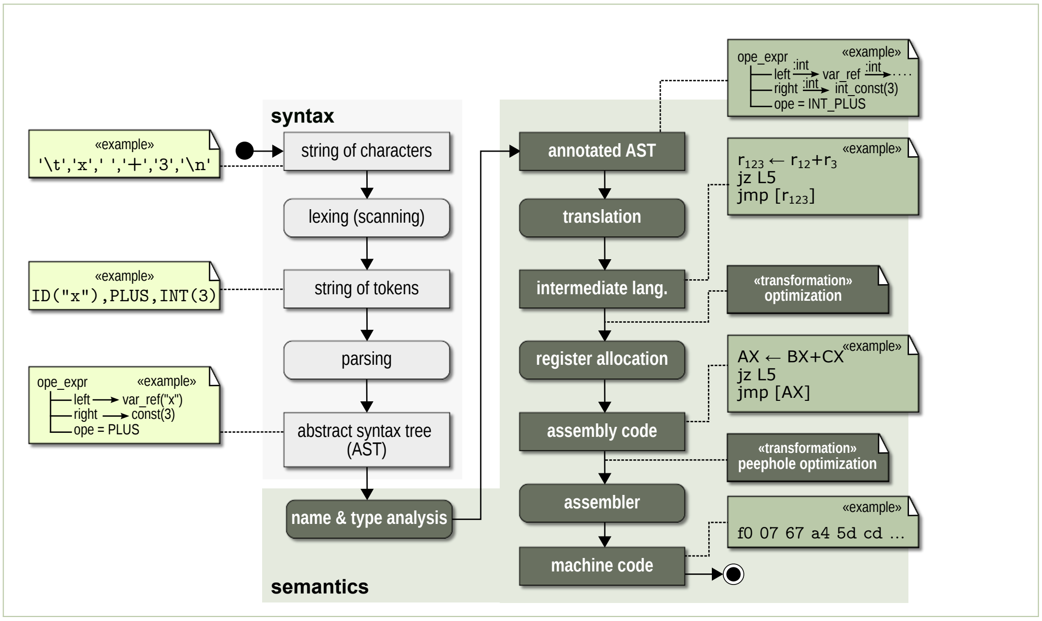


Figure 8.1: *A typical compiler follows a pipeline architecture*

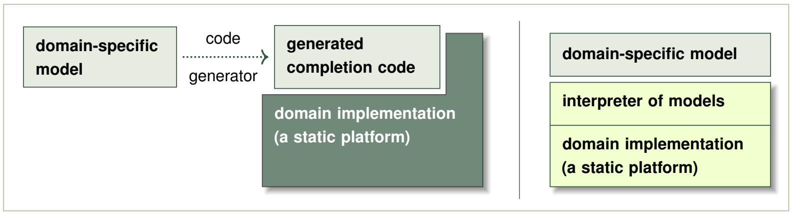

domain-specific model
code generator
generated completion code
domain implementation (a static platform)
domain-specific model
interpreter of models
domain implementation (a static platform)

Figure 8.2: *Left: Code generation creates custom code reflecting the meaning of the model, linked with the platform implementation. Right: An interpreter is a part of the platform. It can use models directly*

The above designs developed for GPLs remain valid for DSLs, but DSLs require flexible and cheap implementation strategies. DSLs are simpler but more diverse than GPLs, and they are developed on smaller budgets. DSL implementers tend to follow the front-end part of the GPL pipeline architecture: the syntactic analysis (Chapters 3 and 4) and the static semantics (Chapters 5 and 6). The multistage code generator passes are often replaced with a simple interpreter, or with a generator targeting a GPL. The complexity is then delegated to the GPL compiler or interpreter.

Figure 8.2 summarizes the difference between these two key strategies: code generation (left) and interpretation (right). With code generation we hard-code a static platform represented by the dark shape in the figure. The static platform implements the part of our DSL that is independent of the model; typically a runtime system, a business application framework, hardware drivers for an embedded system, etc. The static platform should expose abstractions that can be efficiently used by the generated code. Then we generate the remaining code automatically, based on the input model. The generated code reflects the meaning of the model. It is linked and executed together with the static platform to obtain a running software system. Both the static platform and the generated code are implemented in a target GPL. Google Protocol Buffers (p. 10) are an example of a DSL using (multiple) code generators and (multiple) static runtime implementations.

In the interpreter strategy (on the right in Fig. 8.2) we still implement a static platform, but instead of a code generator, we include a runtime interpreter for models. There is no generation whatsoever. The platform just reads the model and executes it with the provided logic. Most block-based languages for programming aiming at children (such as the Scratch language in Fig. 1.3) are interpreted. However, many DSLs for professionals are also interpreted, among them block-languages for configuring robotics applications (similar to Scratch), HTML (where the verb rendering means interpretation), and query languages (like SQL). Interpretation is often the easiest and the cheapest way to implement DSL semantics, thus we discuss it in considerable detail below. Code generation, as a more performant, but also more expensive method, is discussed in Chapter 9.

8.1 Domain Implementation

With both code generation and interpretation (Fig. 8.2), the bulk of any DSL implementation tends to be *model-independent*. One can implement a large part of the language semantics as usual "static" code without building an interpreter or generating code. For example, in a DSL describing web pages, most of the logic can be pre-implemented in generic JavaScript, CSS, and HTML primitives, all independent of the input model. In a mission DSL for robotics, most of the implementation is concerned with robotics architecture, hardware drivers, perception, knowledge, and control components, which all can be implemented and composed regardless of whether we use a model to control a robot or not. Similarly, only a tiny part of a database management system is concerned with interpretation of SQL. Most of the manipulation primitives, data structures, indexing, and storing solutions are implemented before you write your first query. An SQL interpreter merely composes the right elements from this implementation to deliver a response to a query.

This static project, often called the *platform*, implements what is known about the domain, limiting the use of the model to interpretation or code generation time.

A library as the platform. The domain can be implemented as a library, if the DSL's semantics concerns invoking primitive operations from a known collection. This is, for instance, the case for control languages that define top-level behavior of an agent like a robot or another device. Stahl and Völter [17] recommend placing domain concepts at the level of functionality, when implementing the platform. This decreases the semantic gap between models and code, and maximizes the chance of success when translating or interpreting a model. So, for example, in an insurance application it makes sense to work with platform concepts like person or account, and the primitive operations on them (open, close, deposit, index, evaluate, etc.). In a robotics application, it is useful to work with robot actions and skills, for instance: moving, sensing, planning, and following paths for a mobile robot. The types in your library should capture the domain concepts (meta-model types can often be used directly here), and the API functions should reflect the basic operations in the semantics of the DSL. Then the generator or the interpreter just translates the model elements to instances of these types and the model operations to the API calls.

A framework as the platform. If your DSL is not control-oriented, and the same control loop is used in the semantics of all models (for instance in event-driven languages), it might be more practical to implement the platform as a *framework* with control inversion. A framework is a larger code base, typically with quite a strict architecture, that can be extended into a complete application by providing adaptation code (for example via extending super-classes), providing configuration parameters, XML models, or implementing callbacks. You can think of a framework as a large library using *inversion of control*. Frameworks are ubiquitous in software development; we use persistence frameworks, GUI frameworks, web programming frameworks, or enterprise systems frameworks.

Very often DSLs are designed against existing frameworks (Eclipse, OSGI, ROS, etc.). The framework adaptation code tends to contain a lot of boilerplate, which can be successfully generated from models describing the essence of the product. Automatic code generation from models or interpretation of models helps to maintain invariants of the framework API—this is very handy, as violation of such invariants typically cannot be discovered early (errors manifest themselves only at runtime). If the code generator/interpreter is correct, we just need to encode the framework invariants into the static semantics of the DSL by adding suitable constraints or typing rules.

Other good architectural frameworks that work well with MDSE are component-based architectures and middleware, which both require considerable boilerplate code to integrate together into complete products. This code can normally be generated automatically.

Ultimately, you rarely have the comfort of choosing the architecture and technology underlying your DSL. For most commercial software, at the point when a DSL is being built, the underlying software platform already exists. The only thing you might need to do is to add an adaptation layer that brings the abstractions closer to the DSL concepts.

The organization of a DSL implementation into a large platform and a small interpreter (generator) allows the amount of information in the models to be limited. The languages can be kept small, and users need not provide a lot of details in them. The DSL tools extract relatively little information from the models, and mostly focus on composition and completing the platform implementation. The models are easier to write for users and the DSL is easier to implement for engineers.

Example 24. Recall the DSL `robot` of Chapter 2 (Fig. 2.2, p. 30), a state-machine-like language with nested modes containing actions executed on mode activation, and reactions triggered by external events. Figure 8.3 presents its meta-model. Actions (docking, turning, or moving) are not instantaneous but can have a predefined duration, which is specified using a minimalistic expression language. Reactions are predefined state changes triggered by detection of an obstacle or a sound signal (a clap). Finally, a dedicated action navigates back to the initial position of the robot while avoiding collisions.

We chose to create a domain implementation for `robot` using the Robot Operating System (https://www.ros.org/), a robotics programming platform that bundles simulators of the hardware along with sensing and control software. The overall architecture is summarized in Fig. 8.4, to the right.

To execute models like the one in Fig. 2.2, we need a physical robot, but in early stages it is more practical to use a simulator (bottom in the diagram). We use the TurtleBot3 platform along with the Webots simulator (https://cyberbotics.com/). This way you can run and test it without the hardware. The simulator provides a model of the physical environment and of the robot hardware. ROS provides a differential drive controller (that moves the robot by translating velocity commands into power applied to the two wheels) and a driver for the laser sensor to detect obstacles. Finally, for navigating back to the initial position, we use a navigation component from ROS that computes paths on the map and uses the controller to follow them. Besides these, ROS also provides basic coordination and communication middleware so that we can monitor and

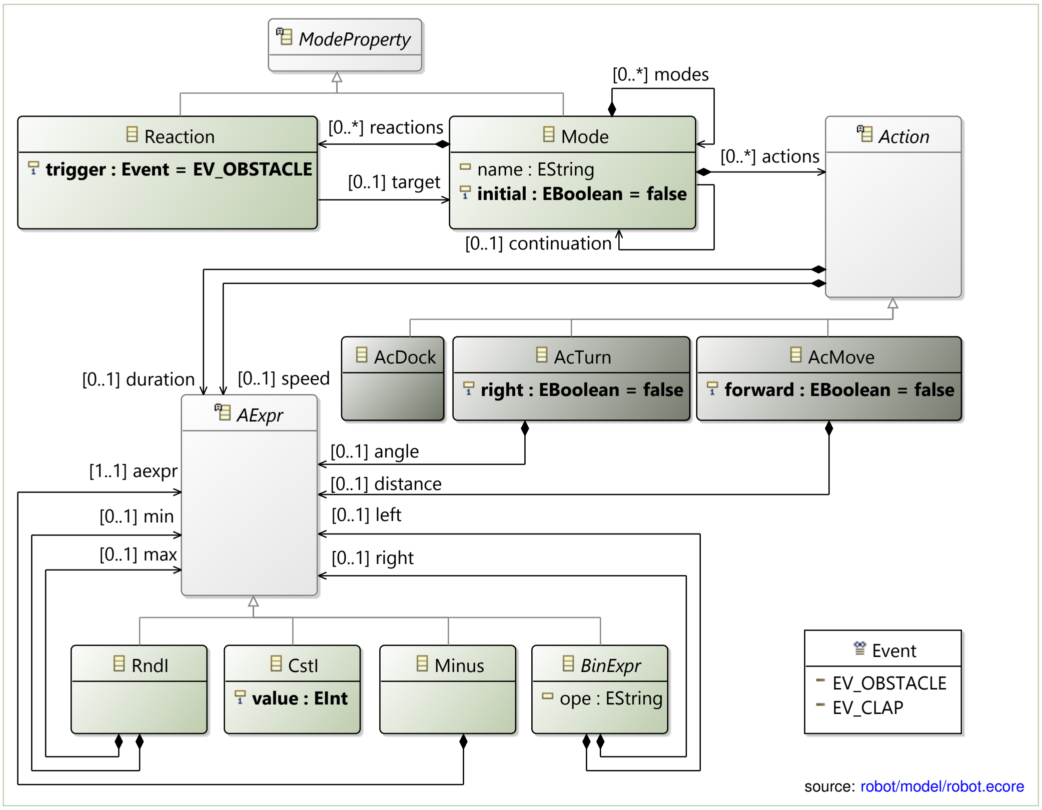


Figure 8.3: A meta-model of a simple control DSL for mobile robots. See an example model in Figures 2.2 and 2.5

control the components of our interpreter. Since the framework is fairly substantial and requires extensive setup and configuration, we wrapped everything in a Docker container. We have done this to simplify the setup for readers, but the practice is not unusual for complex platforms underlying real DSLs—you often need to provide a reproducible development and deployment solution, not only in robotics. For instance, you cannot expect every developer building a financial product DSL to set up the test and build environment involving complex business servers. This is best automated in a reproducible manner.

This book is not about robot programming—the above summary, necessarily too brief to build a robot, has been included to make you realize that the implementation of the domain for many DSLs may be substantial, unlike what many textbook examples may make you think. The platform implementing the domain is typically much larger and much more complex than the actual interpreter or a code generator. The platform is also what gives the DSL its power—a rich semantics for the models.

Often the existing infrastructure predates the DSL design project. For this reason it may turn out not be sufficient, even if correctly configured. A common problem is an impedance between the abstractions in the language

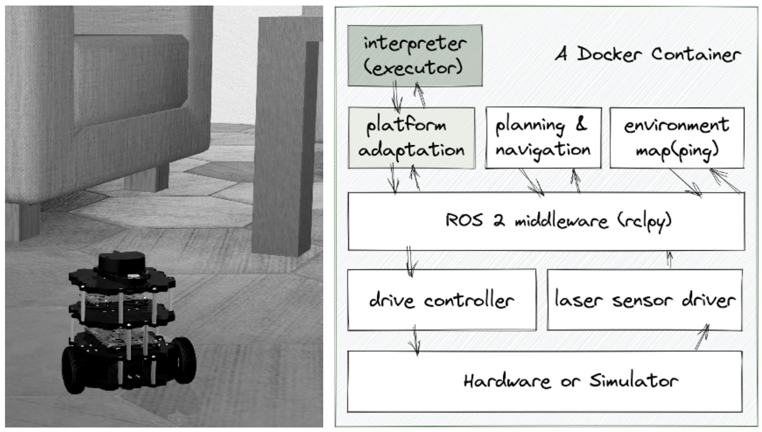


Figure 8.4: *Left: A visualization of the TurtleBot3 robot in a Webots simulation of an example project robot.turtlebot3. Right: The key architectural components realizing the logic of a `robot` model, a refinement of the right-hand-side diagram of Fig. 8.2 for this example. The interpreter is just a single component here, not even the most complex one. The domain implementation tends to be large and complex for many DSLs*

and in the legacy platform. For this reason it is advisable to implement an adaptation API layer that directly corresponds to the primitives in the DSL, but realizes the meaning of the primitives using the platform. This should make the implementation of the interpreter much simpler.

Example 25. For the `robot` language, the adaptation layer defines functions realizing actions on top of the ROS API, and establishes listeners for external events used in the models. It is shown as a gray box in the top right of Fig. 8.4, below the interpreter. Figure 8.5 summarizes the adaptation API. Focus on the kinds of APIs implemented here; most function bodies are elided anyway, for brevity. The class `TurtleBotPlatform` is an abstract class, which is extended by model executors, for instance interpreters, but also reference implementations of models, and generated code. The initializer in lines 2–15 defines the operation frequency, a number of event monitors (callbacks), and several internal *state variables*. The callbacks, l. 7–9, are activated whenever the laser sensor communicates a readout (used to discover obstacles), whenever a user produces a clapping sound, and when a timer-driven loop needs to republish active commands for the motor (l. 9). We create a publisher channel for communicating with the differential driver controller in l. 6. The state variables store the active motor command (l. 11), the current mode of the model (l. 12), and two flags registering whether a clap or obstacle event has been seen and awaits processing (l. 13–14).

The most interesting part of the figure is the action API in l. 31–38 implementing operations available in `robot`. There is almost a one-to-one correspondence between these functions and the actions available in the meta-model of Fig. 2.2. This is a common pattern: the domain implementation API should closely mimic the language. Finally, the functions in lines 17–29 (and in l. 40) implement other technicalities of the `robot` semantics: the monitoring and timing callbacks, and making motor commands persist for some time (latching).[1]

[1] Our code repository also contains a simpler version of this example, implemented in Scala, using RosJava and ROS1, and integrated with xtext. See robot.scala/.

```python
class TurtleBotPlatform(Node):
    def __init__(self, name, executor):
        super().__init__(name)
        self.FREQ = 10.0 # Hz
        # Framework event listeners and timers
        self.cmd_vel = self.create_publisher(Twist, '/cmd_vel', 10)
        self.scan_listener = self.create_subscription(LaserScan,'/scan',self.scanner_callback, 1)
        self.clap_listener = self.create_subscription(Bool, '/clap', self.clap_callback, 10)
        self.tm_control_loop = self.create_timer(1.0/self.FREQ, self.msg_pulse_callback)
        # Internal framework states
        self.latched_msg = Twist()
        self.mode = []
        self.ev_clap = False
        self.ev_obstacle = False
        ...

    def info(self, msg: str) -> None:
        """A convenience function for reporting progress using ROS logging facility."""
    def latch(self, duration: float = -1.0) -> None:
        """'Latch' the message for non-preemptive duration seconds,
           or if duration -1, latch it for infinity but make preemptive."""
    def tm_latch_callback(self) -> None:
        """Discover that a message has timed out, and 'unlatch' it."""
    def msg_pulse_callback(self) -> None:
        """Repeatedly send latched message to /cmd_vel, if there is one."""
    def scanner_callback(self, msg: LaserScan) -> None:
        """Detect an obstacle with reasonable range of sensing in front"""
    def clap_callback(self, msg: Bool) -> None:
        """Called when a clap is observed (mocked)"""

    # Operations on the robot (basic actions semantics)
    def random_rotation(self, duration: float = -1.0) -> None: ...
    def engage(self, velocity: float, duration: float = -0.7) -> None: ...
    def stop(self, duration: float = -1.0) -> None: ...
    def back_off(self, duration: float = 0.7) -> None: ...
    def return_to_base(self) -> None: ...
    def _go_to_pose(self, pose: PoseWithCovarianceStamped) -> None: ...
        """Plan and Navigate to a specific pose on the map using Navigation2 of ROS2"""

def swist(m) -> str: """Format a 2D twist message a string for printing""" ...
def run(args, mkController):
    """Initialize ROS and the interpreter class, start the main control loop""" ...
```

source: robot.turtlebot3/dsldesign_robot_turtlebot3/turtlebot.py

Figure 8.5: *The adaptation layer API (see Fig. 8.4) for the* `robot` *DSL example, in Python using ROS*

In summary, the adaptation layer API implements the basic operations of the language, defines the state of the execution, and provides infrastructure services needed for a language interpreter (or a code generator) to execute models possibly directly. The interpreter does not need to refer to other parts of the platform, as all the functionality is exposed in the adaptation layer at the right level of abstraction. Crucially, the entire platform, even the adaptation API, is independent of the input model. It does not refer to the model.

Discussing domain implementations in detail is difficult, because they entirely depend on expertise in a particular domain of interest. For effi-

```scala
def eval (s: fsm.State) (input: String)
  : Option[(fsm.State, Option[String])] =
  s.getLeavingTransitions
   .asScala
   .find { tran => tran.getInput == input }
   .map { tran =>
     (tran.getTarget, Option (tran.getOutput)) }


def repl (s: fsm.State): Unit =

  val inputs = s.getLeavingTransitions.
    .asScala
    .map { _.getInput }
    .mkString (", ")

  print (s"\nMachine in state: ${s.getName}. ")
  print ("Input [$inputs]? ")
  val input = io.StdIn.readLine
  eval (s) (input) match

    case Some (s1 -> Some (output)) =>
      println (s"Machine outputs: $output")
      repl (s1)
    case Some (s1 -> None) =>
      repl (s1)
    case None =>
      println ("Invalid input!")
      repl (s)


def run (m: fsm.Model): Unit =
  repl (m.getMachines.get (0).getInitial)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/package.scala

```python
def eval(state, input: str):
    active = filter(lambda t: t.input==input,
        state.leavingTransitions)
    tran = next(active, None)
    if tran:
        return (tran.target, tran.output)
    else:
        return None

def repl(state):
    while True:
        name = state.name
        msg1 = f'Machine in state "{name}".'
        inputs = (s.input
            for s in state.leavingTransitions)
        msg2 = ', '.join(inputs)
        try:
            print(msg1, end = '')
            i = input(f' Input [{msg2}]? ')
            result = eval(state, i)
            if result:
                state = result[0]
                if result[1]:
                    print('Machine outputs:',
                            end = '')
                    print(f' {result[1]}')
            else:
                print("Invalid input!")
        except EOFError:
            print("\nInvalid input!")

def run(model):
    repl(model.machines[0].initial)
```

source: fsm.py/interpreter.py

***Figure 8.6:** An interpreter for state machines in Scala (left) and Python (right) using the meta-model of Fig. 3.1*

ciency, we switch attention to simple exercise-like DSLs below, without any accompanying platform. Beware though, that this switch is purely motivated by the need to discuss interpretation principles efficiently and pedagogically. It is not meant to be an admission that the platform is unimportant, or even that writing the interpreter is the major part of the required effort.

8.2 The Interpreter Proper

For many simple DSLs the easiest way to interpret a model is to work directly with its abstract syntax. We show this on two simple examples: the finite-state-machine language and the logical expressions language.

Interpreting a language with explicit states and imperative transitions. The characteristic aspect of the finite-state-machine example DSL is that its execution involves an explicit notion of *state* and *transitions* between states. An interpreter for such a language typically takes the form of a loop (or an equivalent recursion) calculating the new value of the current state. Figure 8.6 shows the interpreters for the `fsm` language in Scala and Python. Both implementations work with the abstract syntax as defined in the meta-model of Fig. 3.1. (The ADT variant would be almost identical.)

The core of the interpreter is the function that for the current state and input calculates the target state and the output (lines 1–8, for both languages). We first find the transitions leaving the current state, then identify active transitions (that are triggered by the current `input`), pick the first active transition, and return a pair: the target state and (possibly) an output. If any of these steps fails (no leaving transitions, no active transition) then we return an error value, `None` in both languages. The output message is also `None`, if the transition produces no output (silent).

Many interpreters maintain a *state* of execution (here the active state of the machine). An evaluation mechanism interprets the current state and interacts with the execution environment (the domain implementation) to create a new state. Often the notion of state is implicit, with a complex transition evaluation function. For instance, the state may be the values assigned to all variables in scope (a dictionary) with each transition given by assignment statements. This has been particularly simple for the finite-state-machine language, because in this language the transitions are explicit in the abstract syntax, and firing them requires no calculation.

Read-evaluate-print-loop (REPL). An evaluation function needs to be wrapped into an execution loop. In our example, the loop is a simple infinite iteration that continues to take inputs from the user (standard input), execute transitions using the evaluator, and communicate the potential outputs along with the active state. Lines 10–33 in Fig. 8.6 show implementations for a REPL in Scala and Python. In Scala, we read an input from the user, evaluate it to get a new state using the `eval` function discussed above, print an output (if present), and move to the new state. The loop is explicit in Python (l. 11). The Scala version uses (tail) recursion to achieve the same effect (l. 24, 26, and 28). Of course, we could have used a while-loop in Scala, too.

Exercise 8.1. Reimplement the REPL in Scala using an explicit while-loop and a variable for storing the current state.

Many modern programming languages offer REPLs, sometimes known as an *interactive shell* or an *interactive interpreter*. REPLs help novice users to learn, and the experienced ones to prototype and debug code. REPLs are also the first step to implementing interfaces to more complex environments, such as Jupyter notebooks. They also allow execution of DSL models as shell scripts on any system following the Unix shebang convention (`#!`).[2]

[2]The shebang convention is used to make script files executable in Unix systems and compatible systems. In order to support it, the grammar of your DSL must accept a hash as a character opening a line comment, at least on the first line in the file. To start an interpreter from within a model file use `#!/path/to/interpreter` in its first line and make it executable. To start an interpreter that already reads the model file some other way, but reads the REPL commands from the current file, use `#!/usr/bin/env -S sh -c '"$1"<"$2"' redirun /path/to/interpreter`.

This possibility is one of the easiest ways to integrate your DSL models with the rest of the system.

If the fsm language had to be used to control an embedded system, the REPL would have to be replaced with reading sensors and actuating hardware, instead of standard input and output streams. If control inversion is used, for instance implemented in a framework, the evaluation function needs to be registered in some event-handlers or callbacks, while the framework provides the loop.

Recursive interpretation of ASTs. Recursion is a natural way to traverse meta-models and ADTs with cycles over containments. This is best demonstrated on the simple expression language expr of Figures 7.12 and 7.13. Recall that expr has binary and unary expressions (and, or, not), named variables, and constants (true/false, in the Scala ADT variant).

The essential difference between the state machines and expressions is the limited nesting of the abstract syntax of fsm models. A model contains state machines, machines contain states, each state may contain transitions. No deeper elements are present. In the expression language an expression can inductively contain another expression, which contains another expression, and so on. Arbitrarily large syntax tree structures can be instances of the expr. Recursion is an ideal way to explore such trees.

The second essential difference between fsm and expr is how they focus on state changes and values. The execution of a finite-state machine modifies the current state of the interpreter. In other words, it produces side effects. The expression language is quite different. For example, the expression $x \wedge y$ *evaluates to* the value *true* for the assignment of variables $\{x \mapsto \mathit{true}, y \mapsto \mathit{true}\}$. The state of the interpreter and the assignment of variables to values remain unchanged during the evaluation. The only result of the evaluation is the value produced. In general, model evaluation often produces both state changes and values, but some languages are more value-heavy than state-heavy.

Let us see how these ideas manifest in a Scala interpreter for the ADT version of expr (Fig. 8.7, left column). In Line 1, we define the type Env (for "environment"), which represents the state of the evaluation. Our state is a map of variable names to values. The state never changes here. Contrast this with the fsm interpreter, which produced a new state value at each step. The expression evaluator (eval) produces a Boolean value (lines 3–4) or fails (Option). The type Boolean is our *value type* while Env is our *state type*.

When building an interpreter, you should have a clear idea what state is manipulated and what values are produced. The state type in an interpreter is the type that represents the state of the execution. The value type is the type that represents the value produced. For larger languages, we can have several state and several value types, as different parts of the model have different meaning. Importantly, one should not confuse the state and value types with the meta-model types. For instance, in expr we have a constructor True that represents the literal true *in the syntax*. This syntax element

```scala
type Env = Map[String, Boolean]

def eval (e: Expression) (env: Env)
  : Option[Value] =
  e match

  case Identifier (name) =>
    env.get (name)

  case AND (l, r) =>
    for
      valL <- eval (l) (env)
      valR <- eval (r) (env)
    yield valL && valR

  case OR (l, r) =>
    for
      valL <- eval (l) (env)
      valR <- eval (r) (env)
    yield valL || valR

  case NOT (e) =>
    for  valE <- eval (e) (env)
    yield ! valE

  case True =>
    Some (true)

  case False =>
    Some (false)
```

source: expr.scala/src/main/scala/dsldesign/expr/scala/adt.scala

```java
static class EvalSwitch
  extends ExprSwitch<Boolean> {

  private final Map<String, Boolean> env;

  public EvalSwitch (Map<String, Boolean> env) {
    super ();
    this.env = env;
  }

  @Override
  public Boolean caseIdentifier (Identifier expr) {
     Boolean result = this.env.get (expr.getName());
     if (result == null)
       throw new
           RuntimeError("Access to undefined variable '"
         + expr.getName() + "'");
     else
       return result;
  }

  public Boolean caseAND (AND expr) {
      Boolean valL = this.doSwitch (expr.getLeft ());
      Boolean valR = this.doSwitch (expr.getRight ());
      return valL && valR;
  }

  public Boolean caseOR (OR expr) {
      Boolean valL = this.doSwitch (expr.getLeft ());
      Boolean valR = this.doSwitch (expr.getRight ());
      return valL || valR;
  }

  public Boolean caseNOT (NOT expr) {
      return !this.doSwitch (expr.getExpr ());
  }

  @Override
  public Boolean defaultCase (EObject ignore)
    throws RuntimeError {
    throw new RuntimeError ("Internal Error. " +
        "Attempted to evaluate object " + ignore);
  }
}

static Boolean eval(Map<String, Boolean> env,
                    Expression expr)
                    throws RuntimeError {
  return new EvalSwitch (env).doSwitch (expr);
}
```

source: expr.java/src/main/java/dsldesign/expr/java/Interpreter.java

Figure 8.7: *An interpreter for the* `expr` *language in Scala using the abstract data types AST of Fig. 7.12 in Scala (left) and using the meta-model of Fig. 7.13 in Java (right)*

evaluates to `true`, which is a value of type `Boolean`, our value type. Things often get a bit subtle on the boundary of abstract syntax and value types, and it often helps to carefully separate these. However, once everything is well understood, you may be able to reuse literal types from the meta-model as values in the interpreter, if you know what you are doing. In this example, using the syntax types for values does not really help. Using `Boolean`, we can delegate operations in our language to Scala operations on the Boolean type, which saves some work, and keeps the interpreter smaller. In the `fsm` example, however, we used a reference to a state object (in the abstract syntax) as the state type, effectively reusing a type from the abstract syntax.

The body of the interpreter (lines 5–42) matches the abstract syntax types, and evaluates the syntax nodes. For instance, when evaluating an expression consisting of a single variable name, we return the value of that variable stored in the state environment (lines 12–13). For a more complex binary expression, say `AND` in lines 22–26, we recursively evaluate the left (`l`) and right (`r`) sub-expressions in the same state (`env`). This gives us two Boolean values—we return their Boolean conjunction as implemented in Scala (`&&`). The `for-yield` wrapping makes sure that if any of the recursive calls returns `None` (fails), so does the evaluation of the entire conjunction. We encourage the reader to study the remaining cases.

Let us turn to the right column in Fig. 8.7, which shows a variation of a visitor pattern implementing an interpreter in Java. This interpreter works against the abstract syntax defined by the meta-model of Fig. 7.13, exploiting the infrastructure generated by Ecore. The evaluator class extends the generated switch class for expressions. The value type, still `Boolean`, is returned by the switch (Line 2). The constructor (Line 6) receives the state of the evaluation (still an environment mapping variable names to Booleans) and stores it in the object field (lines 4 and 8) so that the evaluation cases can access it. The methods handling the individual cases of the abstract syntax follow. In the figure, the cases in the right column are aligned with the cases in the Scala interpreter in the left column to make it easy to compare them.

Consider the case of an expression consisting of a single variable name (lines 12–20). We get the value of the variable from the state environment (Line 13), then we check whether this succeeded and either return the value or throw an exception with an error. The code is a bit more verbose than on the left, because `get` in Scala does error handling automatically. On the other hand, in the Java example we formulate an error message. In the conjunction case (lines 22-26), the recursive call now involves invoking the switch method `doSwitch` on the sub-expressions. This method performs the resolution of types to cases using dynamic dispatch. The final case (lines 38–42) is added for diagnostic purposes. If we have not missed any cases, this method will never be called. (The corresponding check in Scala is done for the pattern matching at compile time.)

Exercise 8.2. The interpreter to the right in Fig. 8.7 does not support constant literals `True` and `False`. Extend the Ecore meta-model for `expr` to support them, like in the ADT of expr.scala/src/main/scala/dsldesign/expr/scala/adt.scala. Regenerate the Ecore code and extend the interpreter in expr.java/src/main/java/dsldesign/expr/java/Interpreter.java to support evaluation for the new literals.

An abstract look at interpretation. Interpreters can be implemented following various architectural patterns. However, before one decides on implementation details, it is useful to agree on the execution semantics (dynamic sematics) of the DSL. This can be done with abstract mathematical notation, which sidesteps the intricacies of implementation patterns. Such notation also makes structural recursion (recursion on the structure of abstract-syntax types) and loops visible, demonstrating that these phenomena are not accidental in our implementation but fundamental to the meaning of languages.

Let us begin with formalizing the dynamic semantics of the `expr` language, the essentials besides the interpreters presented in Fig. 8.7. We will capture the dynamic semantics of `expr` using a ternary evaluation relation. "Ternary" means that this relation binds three elements, in this case: an evaluation state (the environment, ε), a term in abstract syntax (an expression, e), and a produced value (a Boolean, v). We propose the following notation:

$$\langle \overset{\substack{\text{interpreter state}\\\text{(values of variables)}}}{\varepsilon} , \overset{\substack{\text{abstract syntax}\\\text{(an expression in expr)}}}{e} \rangle \to \underset{\substack{\text{value computed by}\\\text{interpreting } e \text{ in } \varepsilon}}{v} \tag{8.1}$$

It is instructive to link the symbols to elements of interpreters in Fig. 8.7: ε corresponds to `env`, e corresponds to the expression `e`, and v is the value returned be the interpreters. We use such judgement (notation) to write the operational evaluation rules for `expr`. In the following, `x` stands for any variable name in an `expr` program, b and b_i stand for Boolean values, e and e_i stand for abstract-syntax trees of expressions. Each rule corresponds to a case in the Scala and Java interpreters for `expr`.

$$(\textsc{Var-Ref})\,\frac{b = \varepsilon(\mathtt{x})}{\langle \varepsilon, \mathtt{x} \rangle \to b} \qquad\qquad (\textsc{Not})\,\frac{\langle \varepsilon, e \rangle \to b_1 \qquad b = \neg b_1}{\langle \varepsilon, \mathtt{!}e \rangle \to b}$$

$$(\textsc{And})\,\frac{\langle \varepsilon, e_1 \rangle \to b_1 \qquad \langle \varepsilon, e_2 \rangle \to b_2 \qquad b = b_1 \wedge b_2}{\langle \varepsilon, e_1 \,\mathtt{\&\&}\, e_2 \rangle \to b}$$

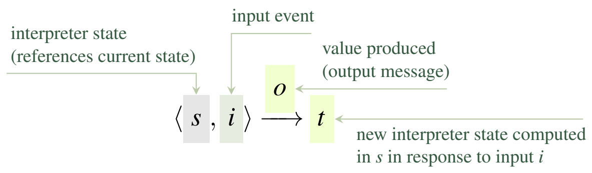
$$(\textsc{Or})\,\frac{\langle \varepsilon, e_1 \rangle \to b_1 \qquad \langle \varepsilon, e_2 \rangle \to b_2 \qquad b = b_1 \vee b_2}{\langle \varepsilon, e_1 \,\mathtt{||}\, e_2 \rangle \to b}$$

$$(\textsc{True})\,\frac{}{\langle \varepsilon, \mathtt{true} \rangle \to \mathit{true}} \qquad\qquad (\textsc{False})\,\frac{}{\langle \varepsilon, \mathtt{false} \rangle \to \mathit{false}}$$

Recall that inference rules are read upwards. The conclusion, under the line, describes the purpose of a rule. The premise, above the line, specifies when the rule applies. For example, VAR-REF defines how to evaluate an expression consisting of a single variable reference `x` to produce a value b. It checks the value of `x` in the environment ε (the premise) and returns it as the result of the evaluation (b) in the conclusion. The negation rule NOT defines how to interpret an expression that negates a smaller sub-expression e (the conclusion). The rule evaluates the expression under negation recursively (the premise) and returns the negation of the obtained Boolean value. We encourage the reader to study the remaining rules.

Exercise 8.3. Compare Fig. 8.7 with the above rules. Explain how the formal rules are realized in the two interpreters in the figure. Draw arrows between rule elements and computations in the source code, while explaining this relationship.

Could we write similar rules for the `fsm` language? This language not only produces values (outputs), but also changes the current state (a side effect). This requires an evaluation relation that binds together quadruples: a source state, an input event, an output, and a target state. Following common convention, we will write the outputted value (o) on top of the evaluation arrow, and the new state (t) to the right of the arrow:


interpreter state
(references current state)
input event
value produced
(output message)
⟨ s , i ⟩ —o→ t
new interpreter state computed
in s in response to input i
(8.2)

Already the structure of the judgement signals that the semantics is concerned with loops: it starts with a state and ends with a new state in each step. This is even more clear in the rules themselves. The rules are not recursive; no premise involves invoking the evaluation judgement as a precondition.

In the following, we write $[s,i,o,t]$ to represent abstract syntax of transitions, where s is the source state, i is the input (activation) label, o is an output label, and t is the target state. If a transition has no output label we write $\perp$ instead. Given a state s we write $s.leavingTransitions$ for the set of transitions sourced in s. In the outputs, we use $\perp$ to mark execution of silent transitions—transitions that do not produce an output.

$$(\textsc{Output})\,\frac{[s,i,o,t] \in s.leavingTransitions \qquad o \text{ is an output label}}{\langle s,i\rangle \xrightarrow{o} t}$$

$$(\textsc{Silent})\,\frac{[s,i,\perp,t] \in s.leavingTransitions}{\langle s,i\rangle \xrightarrow{\perp} t}$$

$$(\textsc{Error})\,\frac{[s,i,o,t] \notin s.leavingTransitions \text{ for any } o \text{ (or } \perp\text{) and any target } t}{\langle s,i\rangle \xrightarrow{\perp} s}$$

The first rule (OUTPUT) identifies a transition leaving state s which is labeled by the presented input i. It outputs the label o found in this transition, and returns its target state as the new current state (t) in the conclusion. The second rule (SILENT) is similar, except that it matches a transition without an output, hence $\perp$, and produces a silent state change to t. The final rule reflects handling erroneous inputs: if an input is provided for which no transition is active, the state does not change, and the user is given an opportunity to feed it with a new input again. This corresponds to a silent loop, if no transition is activated. We encourage the reader to compare these rules to the implementation in Scala and Python presented in Fig. 8.6.

8.3 Case Study: The Robot Language Interpreter

Let us return to the example of the `robot` DSL from Chapter 2, Fig. 2.2. In Sect. 8.1 we have discussed a possible domain implementation for this language using ROS. Let us revisit this case and discuss the implementation of an example interpreter. Our proposal is shown in Fig. 8.8. The presentation is slightly abbreviated—a full version can be found in our code repository. The figure shows four main parts of the interpreter, from the top: an initializer, an expression evaluator, an action executor, and the main loop (`run`).

In line 3, we initialize the super-class, our adaptation API layer as implemented in Fig. 8.5. Then we set up the pyecore framework. The `robot` metamodel is loaded along with an example model to interpret—the random walk model of Fig. 2.2. Finally the top-level mode of the model is made active. The function `activate` switches the current mode to the given one, and initializes the program counter to the number of actions in the current mode.

The core of the interpreter is realized in the loop in lines 31–54. At each iteration of this loop, the interpreter does just one semantic action, either waiting for the current timed actions to terminate (l. 33-34), executing the next action in the current mode (l. 36–37), activating the initial mode nested in the current node if no more actions are to be executed (l. 39), or processing reactions (lines 43–54). The execution of actions uses a helper function `execute_action` (above) that delegates further to our adaptation API from Fig. 8.5. Crucially, the function is so simple because we designed our adaptation layer to provide the right calls for the DSL actions. Not much computation is needed on the interpreter side. The executor also uses a simple evaluator (l. 11). We hope that the you appreciate the similarity of this evaluator to the one for `expr` shown in Fig. 8.7, but now implemented in Python.

Let us return to the interpreter loop. Observe that at every iteration of the loop the interpreter releases control to ROS middleware (l. 32), so that all concurrent aspects of the execution get a chance to schedule. This allows ROS to activate callbacks in our platform which register incoming events and communicate with other parts of the system, including the navigation stack. We used a single-threaded scheduler in this implementation so there is no implicit pre-emption (while the control flow is a bit easier to comprehend). Also, if a command has been issued for a definite duration,

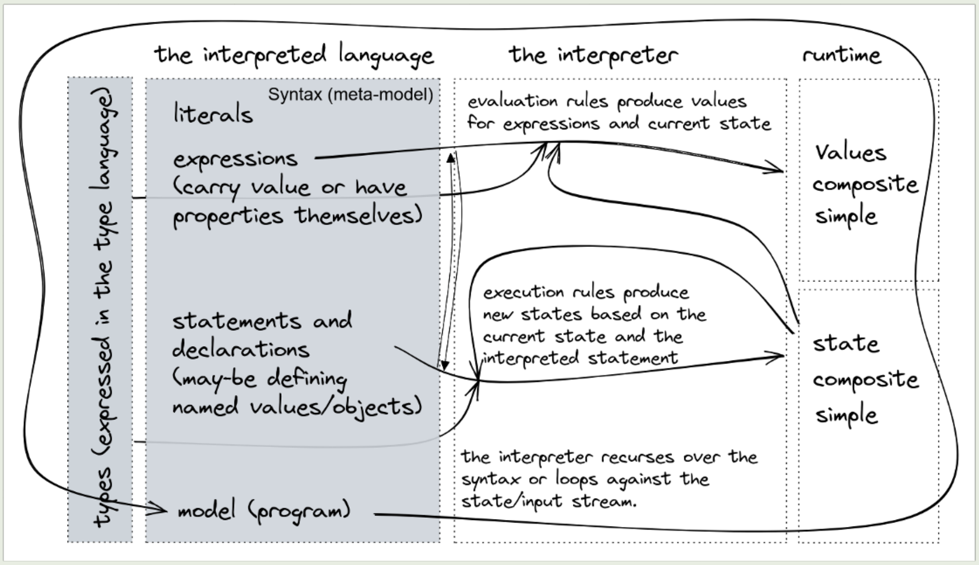
```python
class Interpreter(TurtleBotPlatform):
    def __init__(self, executor):
        super().__init__("interpreter", executor)
        ...
        self.model = self.load_instance("src/dsldesign_robot/test-files/random-walk.robot.xmi")
        self.activate(self.model)

    def activate(self, mode):
        self.mode, self.mode_pc = mode, len(self.mode.actions)

    def evaluate_expr(self, expr):
        if isinstance(expr, self.Robot.RndI): return random.randrange(0, 2000) / 1000.0
        elif isinstance(expr, self.Robot.CstI): return expr.value
        elif isinstance(expr, self.Robot.Minus): return - self.evaluate_expr (expr.aexpr)
        elif isinstance(expr, self.Robot.BinExpr):
            left, right = self.evaluate_expr (expr.left), self.evaluate_expr (expr.right)
            if expr.ope == '+': return left + right
            elif expr.ope == '-': return left - right ...

    def execute_action(self, action):
        if isinstance(action, self.Robot.AcMove):
            direction = 1.0 if action.forward == True else -1.0
            if action.speed: velocity = self.evaluate_expr(action.speed) * 0.008 * direction
            else: velocity = 0.1 * direction
            if action.duration: self.engage(velocity, float(self.evaluate_expr(action.duration)))
            else: self.engage(velocity)
        elif isinstance(action, self.Robot.AcTurn): self.random_rotation()
        elif isinstance(action, self.Robot.AcDock): self.return_to_base()

    def run(self):
        while True:
            self.executor.spin_once(0.0)
            if self.tm_latch in self.timers:
                self.executor.spin_once(1.0/self.FREQ)
            elif self.mode_pc > 0:
                self.execute_action(self.mode.actions[-self.mode_pc])
                self.mode_pc = self.mode_pc - 1
            else:
                try: self.activate(next(filter(lambda m: m.initial, self.mode.modes)))
                except StopIteration:
                    reacted = False
                    for m in self.active_modes():
                        for r in m.reactions:
                            if (self.ev_clap and r.trigger == self.Robot.Event.EV_CLAP) or (
                                self.ev_obstacle and r.trigger == self.Robot.Event.EV_OBSTACLE):
                                if r.target: self.activate(r.target)
                                reacted = True
                                break
                            elif self.mode.continuation:
                                self.activate(self.mode.continuation)
                                reacted = True
                                break
                        if reacted: break
                    self.ev_clap, self.ev_obstacle = False, False
```

source: robot.turtlebot3/dsldesign_robot_turtlebot3/interpreter.py

Figure 8.8: *Core parts of the* `robot` *interpreter for TurtleBot3, slightly abbreviated for ease of reading*

What do I need to build when implementing an interpreter?

Building an interpreter requires connecting different parts of a DSL implementation. We summarize this in the diagram below. The interpreter is typically oblivious to the concrete syntax (thus it is not shown in the diagram), but works with abstract syntax and types (the two gray boxes to the left).


the interpreted language
the interpreter
runtime
types (expressed in the type language)
Syntax (meta-model)
literals
expressions (carry value or have properties themselves)
statements and declarations (may-be defining named values/objects)
model (program)
evaluation rules produce values for expressions and current state
execution rules produce new states based on the current state and the interpreted statement
the interpreter recurses over the syntax or loops against the state/input stream.
Values composite simple
state composite simple

The *abstract syntax* (the interpreted language) is usually the main source of information. The interpreter follows statements/transitions or expressions—these are all elements in a model.

The *value and state types* (rightmost) define what can be computed during interpretation. Values are consumable information; they enter further evaluation or are communicated to the user. States are updated internally by the interpreter. In our examples, the value types were simple (`String`, `Boolean`). The state type for the `expr` interpreter was a map of variable names to values (complex), for `fsm` it was a state name (simple, but we used a reference to a state object in the abstract syntax, which is complex). In general, both value and state types can be either complex or simple, depending on the semantics of the DSL.

The *evaluation rules* (the interpreter proper, the white box in the center) evaluate expressions producing values, or statements producing state changes. In general, if the expressions in a language can have side effects, evaluation of expressions can also produce state changes. For instance, in Java every expression can contain a call to an impure function. Statements often contain expressions, thus statement evaluation rules can depend on expression evaluation rules and vice versa.

The loop around the entire diagram symbolizes scheduling of the evaluation rules. They either follow model elements (for instance statement-by-statement) or the environment inputs, using a recursion, a traversal, a visitor, or a loop. This loop can also be realized by control inversion, when using a framework.

In our examples, DSL *types* were not used, as both languages were untyped. If your DSL has a type checker, the inferred type information may be used in evaluation rules. For instance, many object-oriented languages use static type information to resolve argument overloading or dispatch for non-virtual methods. In DSLs, different types can lead to different semantics as well.

the loop simply sleeps for some time, again releasing control to ROS (l. 34). Unlike a usual interpreter for a programming language, this interpreter works in lockstep and interleaves execution with the operating system in a manner controlled by a clock.

You rarely see interpreters sleeping in programming language textbooks. There is a good reason for this. Regular sources teaching you about implementing programming languages focus on evaluation (like our `expr` and `fsm` examples). Implementing a DSL interpreter requires not only knowledge about languages, but also about the domain, here concurrent and distributed programming with ROS. Thus your DSL interpreters will be heavily influenced by the domain you are targeting. By building the interpreter, you are hiding a lot of complexity in the domain behind the DSL, so that your users do not have to know about all these technicalities. Ultimately, your domain expertise is more important when building an interpreter than what you can learn from this book.

8.4 Monitoring and Models-At-Runtime

Runtime monitoring is an interesting application of interpretation, where the model is not used to run the system by means of an interpreter. Instead, we use the model interpreter to watch the system, for instance to explain its operation or to detect anomalous situations. The system may run using control software developed some other way, not using a DSL or using code generated from a DSL. A runtime monitor is an interpreter that is not concerned with computing values or outputs, but its primary task is to observe the outputs from the system and update the state of the model, based on these observations. In this sense, it resembles our `fsm` interpreter, but treating system outputs as model inputs. The interpreter may raise alarms to the operator, when anomalies are detected. A user interface may be provided which allows the user to benefit from the received information.

Blair, Bencomo, and France [3] survey the basic definitions and applications of models-at-runtime, which is a broader area that incorporates use of models in systems during operations, not only for monitoring.[3] In this domain, the models are system representations, casually connected to the system: either the model controls the system, or the system controls the model (monitoring and explaining). Both of these directions are combined in self-adaptive systems, where the model monitors information about the state of the system and environment, and uses this to calculate adaptation decisions.

Another important application of domain-specific models is adaptation and customization of applications at runtime, after deployment. Common examples include report generation in business applications, and business process models used to customize case-handling systems [16]. Changing models, often stored in databases or standardized formats (XML, Json, Yaml), is much easier than updating the source code of systems. Adaptation by changing models is often available to non-programmers.

[3] See also other articles on this topic in this special issue of IEEE Computer.

8.5 Guidelines for Implementing DSL Interpreters

Let's discuss common advise for implementing interpreters. Be sure to also check the guidelines in Chapter 9, especially regarding the choice between code generation and interpretation.

Use a reference example implementation to design the platform API. You may wonder how did we arrive at the adaptation layer API for the robot example? The domain operations named in the meta-model helped (move forward, rotate, dock), however they did not tell us much about organizing the control structure of the platform. What callbacks do we need, what scheduling policy, and how do we avoid data races and deadlocks for the `robot` language? **Guideline 8.1**

A common practice to help the design process at this stage is to create a *reference product implementation*—a complete manual implementation of a simple but non-trivial DSL model. For our `robot` DSL we first implemented the random walk program directly in Python. You can use any way you choose to do this, but it is most useful to structure the implementation similarly to the abstract syntax of the model. Write a program which behaves like the model, but does not refer to it in the code. Just keep the abstract syntax in front of you, developing from one step to another as if you were the interpreter. Ask yourself: what information would be available at this stage, if I had to take it from the model?

Once this single example is working, refactor the code to keep model-dependent parts in one module and all the model-independent parts elsewhere. This way you get the first platform API candidate. For our robotics example the platform API can be found in robot.turtlebot3/dsldesign_robot_turtlebot3/turtlebot.py and the model-dependent part of the reference implementation is in robot.turtlebot3/dsldesign_robot_turtlebot3/controller.py. We will also discuss it further in Sect. 9.5.

Often you can throw out the reference implementation after building the interpreter. During the interpreter implementation you will find issues with the platform. It should be further developed and structured. The reference example will quickly become obsolete. Its role is fulfilled once you have the platform, and know how to start on the interpreter. In Chapter 9, we show how the reference example helps to implement a code generator; there we also discuss the example for the `robot` language in more detail.

Align the domain implementation and the runtime representation with transformations. Another way to decrease the gap between the underlying software platform and the abstraction level of your DSL is to switch to a lower-level representation before interpreting. Programming language implementation textbooks tell us that a translator can be made simpler if a better-aligned representation of the source is used. It can be produced using a transformation (Chapter 7). In fact, a classical compiler has several translation phases before machine code is emitted; see Fig. 8.1. Common transformations involve syntactic sugar elimination, switching **Guideline 8.2** to an abstract representation of the execution platform (abstract registers instead of program variables), and optimization passes.

Multiple transformation steps are possible also for DSLs, however for most DSLs, we want to keep the back-end simple. It is typically easier to reduce the abstraction gap by developing a suitable domain adaptation layer API than by aggressively transforming the input DSL (besides the simplest desugaring steps during parsing). Static code is much easier to test and maintain than transformations. We especially discourage adding transformations solely to improve performance early, before you have a prototype interpreter working. Optimizations can always be done later.

Guideline 8.3 *Do not confuse model syntax with runtime state.* Meta-models should be used purely to define syntax, and not runtime state. For inexperienced language designers, it is often confusing to decide what information is part of the model, and what a part of its interpretation. Let's discuss this using the examples of `fsm` and `petrinet`.

In the `fsm` interpreters in Fig. 8.6, we use a variable to represent the current state: `s` in Scala and `state` in Python. In both cases, the variable *refers to* the meta-model to point to the active state but itself it is not a part of the meta-model but of the interpreter. We initialize it with the property `initial` of the `FiniteStateMachine` class. We could have placed a reference to the current state, say `currentState`, as a property in the `Model` class (see Fig. 3.1, p. 53). If we did this, we would have to modify (rewrite) the syntax of the model during execution, every time a state changes.

In the `petrinet` meta-model (Fig. 7.3, p. 239), we store the number of tokens in the `tokenNo` attribute of the `Place` class. The role of `tokenNo` is to specify how many tokens are in each place in the *initial* configuration. One could implement an interpreter for `petrinet` by modifying the `tokenNo` values during execution, however we recommend storing the runtime information elsewhere, in a map data structure or an array in the interpreter.

Polluting meta-models with runtime information will confuse the later maintainers of your project, and when you start to use your models for other purposes than execution (for instance visualization). Suddenly, instances have parts that become irrelevant, or they lack information used by your additional back-ends. Every time you create a new back-end for your DSL, you will have to make changes to the meta-model. Since a meta-model is a pivotal central artifact, this can have unpleasant ripple effects.

For the same reasons, we advocate not to place operations on meta-classes. The interpreter or code generation logic is cleaner to keep outside of the meta-models and the code generated from them. Let the abstract-syntax specification be their only function. You can use syntax as runtime representation, but do not embed runtime representation into syntax. *If* your use case requires adding methods to the classes, say because you expose an API to other programmers and they expect object-oriented style, then we recommend to inject this functionality separately. Use extension methods or other injection mechanisms that do not require changing the generated code.

Flavours of Dynamic Semantics for DSLs

When implementing semantics, it is useful to understand what class of language are you dealing with. What languages are similar? How are they built? Find inspiration in this incomplete taxonomy of DSLs.

flavour | examples	description	implementation hints
structural | class diagrams, Ecore, Alloy, Clafer [1], feature models, kconfig, ER diagrams, CSS, protocol buffers (Fig. 1.4), active record (Fig. 1.5), XML	Describe structures of elements and attributes of elements for software and other systems. They range from simple feature models (defining structure of a configuration space) to expressive languages like Alloy [10]. Their semantics defines a set of valid instances.	Rarely interpreted, most often used to generate code or synthesize instances (cf. constraint semantics). Validation of instances is an example of interpretation; done by evaluating constraints similarly to how we did for `expr` but using an instance instead of an environment as a context.
expression (value semantics) | `expr`, `prpro`, OCL, ...	Define expressions computing logical/arithmetic/etc. values (cf. data flow). Used to express guards, policy conditions etc., in larger DSLs.	Their syntax defines trees of terms that can be evaluated. Process them using structural recursion as we have done in Fig. 8.7 for `expr`.
state machines | `fsm`, `robot`, UML state diagrams, Simulink state diagrams, behavior trees [7]	A visible concept of state, one state active at a time. Execute in steps performing state transitions reacting to external or internal stimuli. Some allow concurrency (multiple states active) but at each transition only one state changes (interleaving).	Identify the type of state values, and conditions for the transition relation. The interpreter typically loops to check whether conditions for a transition are satisfied and triggers the required state change, like we did for `fsm` and `robot` in this chapter.
Petri nets | `petrinet`, BPMN, BPL, UML Activity Diagrams	A visible concept of state, multiple states active simultaneously. State changes are concurrent without interleaving. Parallelism not explicit in the structure, like in state machines, but represented at runtime by tokens.	Resembles interpreting many copies of a state machine. An executor maintains a vector of tokens for places (or a database for slow processes). Events trigger change of the placement of tokens.
data flow | spreadsheets, flow charts, hardware circuits, neural networks in forward mode, `prpro`, KNIME (Fig. 1.2), UI DSLs [5]	Models produce value(s) in response to values received. The outcome is calculated by combining values along edges like in expression trees, each edge representing a calculation transforming a value.	Similar to `expr` but the model is not a tree. An interpreter constructs a DAG and pushes values through. For a deep graph of a recurrent model, one can execute calculations in parallel across layers (pipelining).
time-triggered | `robot`, Lustre [8], behavior trees [7]	State-machine/ Petri-net/ data-flow DSLs executed in lockstep against a clock at fixed or variable rate.	Use an operating system scheduler, like for `robot` in Fig. 8.8; run the loop in time slices, not at full speed.
constraint semantics | OCL, Alloy, Clafer, feature models	Logical constraints define a solution space, structural or behavioral.	Best implemented by generating input for a solver (or an M2M transf.)
markup | HTML, XML, Markdown	Languages which produce decorations/properties for otherwise free-style text or similar data. Interpretation of these languages is typically called "rendering."	Resembles generation more than interpretation. Create output in another language, say PostScript, or runtime objects representing the document (DOM, an M2M transformation).

Exercise 8.4. Implement an interpreter for `petrinet`. Use a dictionary or a map to store the number of elements in each location. Then change it to update the value of `tokenNo` fields in the abstract syntax on every execution step. Reflect on the merits of both methods.

Guideline 8.4 *Dynamic typing is a simple substitute for static constraints and types.* If you have not implemented static semantics, but still want to avoid the interpreter crashing in uncontrollable ways, you can insert dynamic type-checking into the evaluator. Essentially, every time in the evaluator where a type error could appear, first check the types of values received, and fail gracefully (or recover) if the types do not agree.

Exercise 8.5. A type checker for `expr` would have ensured safety of name references, so that we cannot interpret expressions which refer to variables not defined in the environment. Presently, without a type checker, the interpreter will fail at Line 13 (Fig. 8.7, Scala). Modify the interpreter to check whether the name is defined, and produce an error message instead of crashing, when a variable name is not known. The easiest way to do it is probably to throw an exception, like the Java variant does. Alternatively, you can print a message and return `None`. Ultimately, one should change the return type from `Option[Boolean]` to `Either[String, Boolean]` and return the error message on failure.

Of course, similarly one could detect type mismatches and other static errors dynamically.

8.6 Quality Assurance and Testing for Interpreters

Testing of dynamic semantics involves testing the domain implementation and testing the interpreter (or the code generator). This is also how difficulty distributes: testing the domain implementation is easier than testing the interpreter, which in turn is easier than testing a code generator.

Testing the domain implementation. The static code in the platform is testable using standard unit testing and mocking methods. This is an important benefit of moving most of the implementation of the semantics into the platform. We can write tests for the platform elements without considering any input models.

Testing the interpreter. Interpreters are special cases of transformations—it is sometimes hard to think about them the same way, but very often they use the same kind of queries and model access as transformations. This is even more obvious when we consider testing. To test interpreters we need to use examples of statically correct models, or generators of statically correct models, and oracles for properties we want to test. For evaluators, the oracles check whether the value produced exhibits desirable properties; for executors we check properties of the states reached. Most of the advice on testing from Sect. 7.8 applies directly here, including the coverage criteria and randomization strategies.

It is extremely useful to keep the interpreter automatically testable. Among other things this allows the use of randomized automated testing

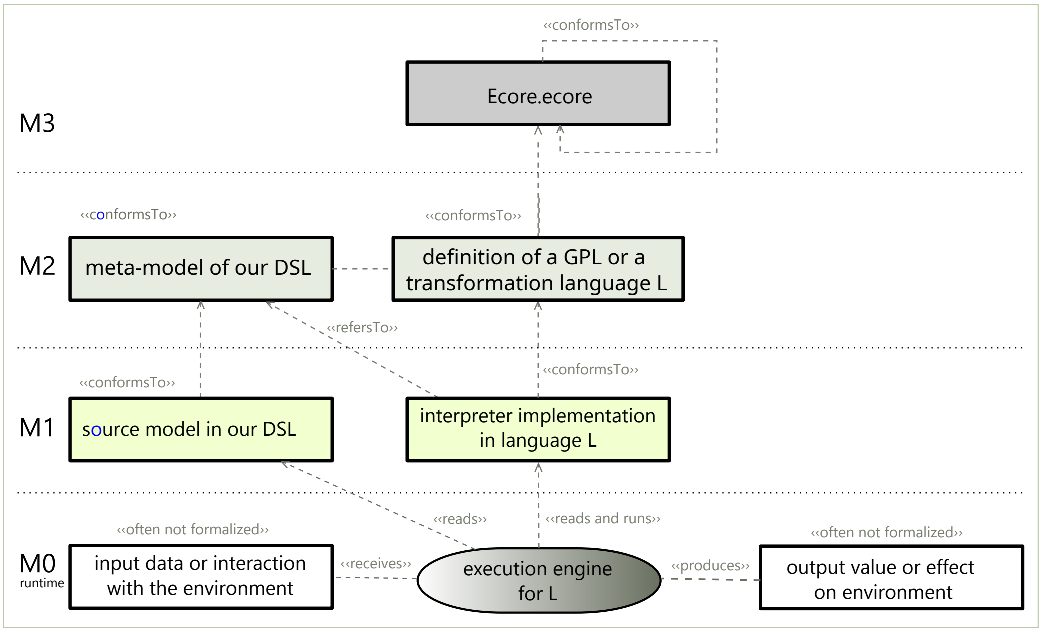


Figure 8.9: Most interpreters are like transformations which do not produce an output at the model level, but at a runtime level

methods like fuzzing, mutation testing, or property-based testing. If the interpreter relies on an external platform (other systems, user inputs, sensor data, etc.) then implement a mock version of the platform, which takes over all communication with the model. This way your interpreter will be testable, without the entire machinery. For instance, it would be very useful to be able to execute basic tests of the `robot` interpreter without running ROS and the physical (or simulated) turtle bot.

Exercise 8.6. Reimplement robot.turtlebot3/dsldesign_robot_turtlebot3/turtlebot.py without any dependency on ROS, as a mock class, and use it to create some test executions of the interpreter of `robot`.

8.7 Interpreters in the Language-Conformance Hierarchy

Figure 8.9 places interpretation in the framework of the language-conformance hierarchy. Since interpretation is a runtime phenomenon the most interesting part is found in the bottom of the figure, at M0. An execution engine running the interpreter reads inputs and produces outputs. The input and the output effect of the interpretation is most often not formalized as a document, thus we mark no meta-models for the white boxes. The input and output values typically conform to a runtime type in the execution language used to implement the interpreter (here denoted `L`).

For `expr` the input was an environment mapping defining variable values; for `fsm` we receive input messages from the operator of our REPL. For `expr` we produced Boolean values; for `prpro` we can produce samples from joint probability distribution (Exercise 8.14); for `fsm` we produced `String` values containing produced messages. However, for `robot` this was much less clear. We did receive and produce messages conforming to ROS message types—still, the motion of the robot was the interesting effect ultimately. Outputs of interpretation can be formalized as documents conforming to formal definitions sometimes, for example, if the interpreter communicates with a web service using documents as messages.

At level M1, in the left column, we have our domain-specific model, conforming to its abstract syntax, at M2. The abstract syntax is itself specified in Ecore (M3) in this diagram. As always, we know that it can be implemented using other means, for instance algebraic data types, or XML schemas.

The interpreter is in the center of the figure, at M1. We assume it has been implemented in a language `L`. All interpreters in this chapter are implemented using GPLs, but `L` could also be a specialized language for meta-programming like PLT-Redex [6] or a model-transformation language (Chapter 7). Imperative transformation languages allow implementation of interpreters in similar style as we did. Pure (graph-rewriting) transformation languages allow implementing interpreters in rewriting style (which we did not use in this book). Again, the implementation GPL `L` does not need to have an explicit meta-model in Ecore. None of our examples did, but model-transformation languages typically have such a meta-model. We draw an Ecore box at the top for simplicity, but of course many implementation GPLs use other defining methods than Ecore.

For all but very few DSLs, the implementation language `L` is different from the DSL itself. This is in stark contrast with the practice of bootstrapping used to implement GPLs: for a GPL it is typically a goal to obtain an interpreter or a compiler implemented in itself, and become independent of other programming languages. Most DSLs are not expressive enough to achieve this. Their goal is also different. Most DSLs want to realize applications in other domains than language implementation, so they are not geared for this problem. With DSLs we use whatever language is convenient to implement the interpreter in. Very often it is the language in which the software platform underlying the DSL is implemented.

Further Reading

Lämmel [12] gives a slightly deeper, but still very approachable introduction to interpretation and the formal operational semantic rules. He has a tendency to focus on classic languages (`expr` rather than `robot`). Further material on formal understanding of the semantics can be found in the books of Nielson and Nielson [14] and of Winskel [18]; both focus on traditional programming languages (not DSLs) and are listed here in increasing level of difficulty. A simple introduction to interpreters and code generators for GPLs, using simple abstract-syntax representations like we do here, can be found in the text book of Sestoft [15].

We have shown two main interpreter implementation patterns: one based on pattern matching and splitting cases, and one on a switch pattern, a variation of a visitor pattern. In object-oriented languages, the *interpreter pattern* is another solution. The interpreter pattern assumes that the evaluation method is implemented by all abstract-syntax classes. It only applies to expression evaluators, and it is hard to use it for DSLs of other semantic styles. It is inconvenient to use with Ecore and generators of meta-classes. If you do not use Ecore, but just implement a syntax ADT yourself, you still mix the evaluation and syntax in complex ways, making your project more entangled. Hills et al. [9] give an informative comparison of the visitor pattern vs the interpreter pattern precisely for this application: implementing evaluators. They conclude that in their experience visitor-pattern-based implementations are cheaper to maintain. Unfortunately, we are not aware of any study that systematically compares the visitor pattern vs a direct recursive implementation with pattern matching (as used in Fig. 8.7).

Combemale et al. [4] give an elegant example of building a type checker using OCL and extend it to an interpreter. The connection between type-checking and interpretation is quite deep. Indeed, type-checking is an abstract form of interpretation.

Our language-conformance diagrams (Fig. 8.9) increasingly focus on transformation and execution, not just conformance. Traditionally, transformation flows have been represented using so-called tombstone diagrams [13]. They focus more on translation and execution than on conformance between languages. You can find a good introduction in the book of Jones, Gomard, and Sestoft [11].

Additional Exercises

Exercise 8.7. Add an implication operator $e_1 \rightarrow e_2$ to `expr` (either in Java or Scala), and extend the interpreter accordingly. The result of $e_1 \rightarrow e_2$ is true if and only if e_1 evaluates to false or e_2 evaluates to true. A similar extension can be done for other logical operators: NAND and XOR (Compare with Exercise 7.3 on p. 263.)

Exercise 8.8. Implement the interpreter for `expr` in Python. For simplicity, you can assume that the expressions are parsed and stored in xmi files, so you can load them using pyecore (https://github.com/pyecore/pyecore). In Python, it is natural to use direct recursion (as Ecore generates no switch class for Python). So we recommend to follow the style of our Scala implementation in Fig. 8.7 but using exceptions to raise errors like in our Java implementation.

Exercise 8.9. Complete the interpreter for `expr` by adding a parser to the available Java interpreter (Fig. 8.7). Use the Xtext parser from expr.xtext/src/main/java/dsldesign/expr/xtext/Expr.xtext following the example in expr.xtext.scala/src/main/scala/dsldesign/expr/xtext/scala/xtextParserExampleMain.scala. If you want to use the Scala interpreter, you need to implement a parser in Scala, or a model transformation that translates the meta-model instance produced by the Xtext parser to the ADT representation used by the interpreter. You should be able to read expressions from a character stream and output their values.

Exercise 8.10. Wrap the interpreter from Exercise 8.9 into a simple REPL, which receives expressions and prints their values. For simplicity, assume a fixed set of predefined variables, all defined to be false in the state environment. The interpreter should fail if you access an undefined variable (as it already does).

Exercise 8.11. Extend the `expr` language to allow definition of variable bindings, for instance by writing: `let x = true`. Extend the abstract-syntax model, the parser, and the evaluator. Now the interpreter either executes a binding, or evaluates an expression. Use this new interpreter in the REPL from Exercise 8.10, which receives expressions and prints their values, but now should also allow definition of variable values.

Exercise 8.12. Study the constant propagation (Fig. 7.14) and expression simplification (Fig. 7.15) presented in Chapter 7. Both of these calculations are forms of *partial evaluation* [11] attempting to evaluate possibly large fragments of an expression with information available statically, without a concrete initial evaluation state. Compare them with the total evaluator in Fig. 8.7. Identify the cases in the code where the partial evaluators give up (cannot improve more), while the total evaluator still proceeds.

Exercise 8.13. Compare the type-checking rules (CONST, VAR-REF, BEXPR) for `prpro` on p. 216 with the evaluation rules for `expr` on p. 305. Observe that the judgements have different notation but similar structure. Both chapters used environments (respectively Γ and ε). What was stored in the type environments, and what was stored in the evaluation environments? Can you explain why type checking is sometimes described as abstract interpretation?

Exercise 8.14. Implement an interpreter for `prpro` as defined in the meta-model of Fig. 6.3 (then use Java or Python) or Fig. 6.2 (then use Scala). The intended semantics is to evaluate expressions like in `expr` and add them to the environments for bindings. A distribution expression should be evaluated by picking a random value from the distribution. For this you may need a statistics library. For Python we recommend `scipy.stats` (https://docs.scipy.org/doc/scipy/reference/stats.html). For Java and Scala, Apache Commons Mathematics Library might do the job (https://commons.apache.org/proper/commons-math/), although it is not well aligned with the pure programming style. As usual, you may need to do some adaptation between the DSL and the underlying platform. Execute each model 20 times, and print out the valuation for all variables at the exit times (a 20-row sample).

Remark: Add a REPL and a few more months of work, and you have developed a mini competitor for R (https://www.r-project.org/about.html).

Exercise 8.15. The same switch pattern that we used in Fig. 8.7 was also used to perform type-checking of the `prpro` models (see Fig. 6.13). It is instructive to compare the two implementations, even though the subject languages differ slightly. Compare the case of type-checking `BExpr` in `prpro` with the case of evaluating `AND` for `expr`. Similarly, compare the case of type-checking `BExpr` in Fig. 6.12 against the case of Scala evaluation of `AND` in the left part of Fig. 8.7. Note how a type checker resembles an evaluator but computes with types not values. What is the essential difference between type-checking and evaluation regarding how the results of the recursive calls are combined?

Exercise 8.16. Note that our implementation of `robot` always performs a fixed random rotation, ignoring the duration and speed of the action provided in the model. Extend the platform and the interpreter to allow rotation with a calculated speed for a calculated time. This exercise requires introductory-level expertise with ROS to produce new `Twist` messages.

Exercise 8.17. Note that our implementation of `robot` always performs a fixed random rotation, ignoring the value of the angle provided in the model. (Open the meta-model from the repository, as Fig. 8.3 does not show the respective references). Add support for rotation by an angle to the platform and the interpreter.

Warning: this requires experience with ROS navigation. A similar extension would add an ability to move forward for a certain distance (presently only time is supported).

Exercise 8.18. Implement a runtime monitor for `robot` that listens to the running interpreter of a given model and shows what mode of the model is active. Of course, in this case it is easier to make the interpreter report the mode changes (it already does)—however for the sake of the exercise, make the interpreter publish mode changes as messages on a new topic (say `/monitor`) and write another ROS node that interprets these messages to report about the state of the system. This exercise requires introductory-level expertise with ROS to set up new channels, add a new node to a project, etc.

Exercise 8.19. Discuss specializations of the diagram in Fig. 8.9 for `expr`, `fsm`, `robot`, and `prpro` (if you solved Exercise 8.14). What goes into the individual boxes?

Exercise 8.20. Study the Ecore meta-model in spreadsheet/model/spreadsheet.ecore and design a simple evaluator for instances of spreadsheets (instances of this meta-model). For simplicity, an evaluator could print an ASCII table or a CSV file with the results. Depending on how much you want to implement, and how much you want to extend the meta-model this project can become arbitrarily large.

References

[1] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wąsowski. “Feature and meta-models in Clafer: Mixed, specialized, and coupled”. In: *International Conference on Software Language Engineering (SLE)*. 2010 (cit. p. 313).

[2] Stanisław Barańczak. “Mały, lecz maksymalistyczny manifest translatologiczny (a small but maximalist translatological manifesto)”. Trans. by Antonia Lloyd-Jones. In: *Teksty Drugie: Teoria Literatury, Krytyka, Interpretacja* 1990.3 (1990) (cit. p. 293).

[3] Gordon Blair, Nelly Bencomo, and Robert B France. “Models@run.time”. In: *Computer* 42.10 (2009), pp. 22–27 (cit. p. 310).

[4] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge Into Tools*. CRC Press, 2016 (cit. p. 317).

[5] Martin Elsman and Anders Schack-Nielsen. “Typelets: A rule-based evaluation model for dynamic, statically typed user interfaces”. In: *Practical Aspects of Declarative Languages (PADL)*. Ed. by Matthew Flatt and Hai-Feng Guo. PADL. Springer, 2014 (cit. p. 313).

[6] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009 (cit. p. 316).

[7] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. “Behavior trees in action: A study of robotics applications”. In: *International Conference on Software Language Engineering (SLE)*. Ed. by Ralf Lämmel, Laurence Tratt, and Juan de Lara. ACM, 2020 (cit. p. 313).

[8] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proc. IEEE* 79.9 (1991), pp. 1305–1320 (cit. p. 313).

[9] Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "A case of visitor versus interpreter pattern". In: *Objects, Models, Components, Patterns – 49th International Conference (TOOLS)*. Ed. by Judith Bishop and Antonio Vallecillo. Springer, 2011 (cit. p. 317).

[10] Daniel Jackson. *Software Abstractions*. MIT Press, 2006 (cit. p. 313).

[11] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Inc., 1993 (cit. pp. 317, 318).

[12] Ralf Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018 (cit. p. 316).

[13] W.M. McKeeman and et al. *A Compiler Generator*. Prentice Hall, 1970 (cit. p. 317).

[14] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications. An Appetizer*. Springer-Verlag, 2007 (cit. p. 316).

[15] Peter Sestoft. *Programming Language Concepts*. Springer Science & Business Media, 2012 (cit. p. 316).

[16] Tijs Slaats, Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Morten Marquard. "Exformatics declarative case management workflows as DCR graphs". In: *Business Process Management (BPM)*. Ed. by Florian Daniel, Jianmin Wang, and Barbara Weber. Springer, 2013 (cit. p. 310).

[17] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005 (cit. p. 295).

[18] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993 (cit. p. 316).

A compiler is much like an interpreter, both in its structure and in the function it performs.
Abelson et al. [1]

9 Code Generation

Even though building interpreters is often the cheapest and the easiest way to implement dynamic semantics, we need alternatives when architectural or performance requirements rule that out. Demands on execution speed, throughput, parallelization, low memory consumption, access locality, security or available programming languages and libraries may all prohibit using an interpreter to execute DSL models. Code generation shifts some of the heavy requirements of the interpretation to compile time, allowing use of simpler target languages and simpler runtime architectures, and delivering higher performance. It helps in the protection of intellectual property, if the input models should not be disclosed. At the same time, code generators tend to be more expensive to implement and test than interpreters.

Code generation is a form of model-to-text (M2T) transformation. There are three popular patterns for implementing code generators: (i) visitor-based or recursive, driven by the input structure, (ii) template-based, driven by the output structure, and (iii) hybrid. Visitor-based generators use the object-oriented visitor pattern to produce output for each element of an AST. A similar solution is to use recursion, often preferred in functional programming languages. Template-based generators take the perspective of the output program, which is written out in full, except for a few gaps left to be filled in by the generator. For more complex DSLs, a hybrid strategy is useful, combining a template with algorithmic traversals of the input AST. In the following, we present these strategies in detail. Subsequently, we discuss design guidelines and quality assurance practices for generators.

9.1 Reference Example Implementation

In Chapters 7 and 8 we used the language `expr` for transformation and interpretation. We have even built a simplistic code generator for it, disguised as the `toString` method in Fig. 7.12 on p. 261. If you review that example, you will realize that a translation of `expr` to a high-level target is almost trivial. Logical expressions in high-level programming languages are all alike, so there is not much to translate (but see Exercise 9.22). We need a more involved example to demonstrate code generation.

Example 26. Recall the simple probabilistic modeling language, `prpro`, used in Chapter 6. In `prpro`, values of variables are defined using expressions involving probability density functions. The density's parameters can themselves be defined by expressions. Data sets can be loaded and used to do inference in

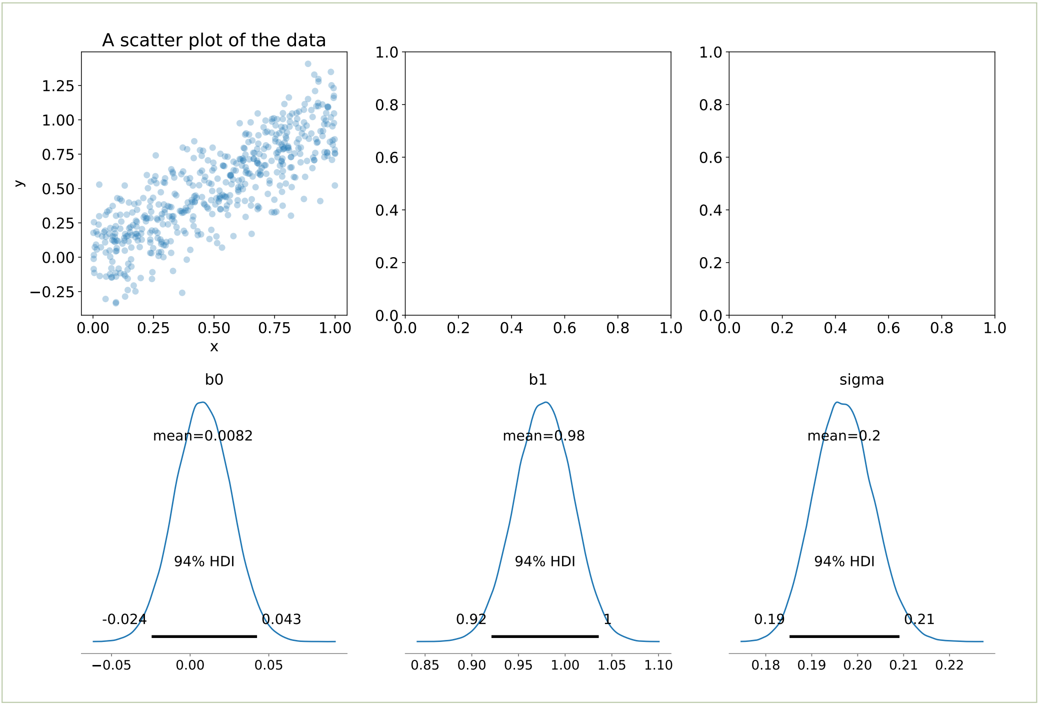


Figure 9.1: An example result of executing code generated from a `prpro` *model. The program builds an inference model and construct the plots (all labels, and the number of plots are extracted from the input). The statistics involved is not of interest to us in this book*

the model. The example model below represents a linear regression problem. We have two data columns, x and y. Variable y is normally distributed (the last line), with the mean given by a linear function of x. The parameters of the function are unknown, but uniformly distributed (b_0, b_1), and so is y's standard deviation σ. The example can also be found at prpro/test-files/example2.xmi.

$$
\begin{aligned}
&\textbf{data } x\\
&\textbf{data } y\\
&b_0 = \mathscr{U}(-2,+2)\\
&b_1 = \mathscr{U}(-2,+2)\\
&\sigma = \mathscr{U}(0.001, 2.0)\\
&y \sim \mathscr{N}(b_1 x + b_0, \sigma)
\end{aligned}
$$

We would like to use `prpro` models to read data and to infer distributions over the parameters b_0, b_1, and σ. Figure 9.1 shows a possible result: the two columns of data against each other (the top left cell), the inferred distributions over the parameters in the bottom row. How do we create such plots using a code generator? We need to be able to create the code like that in Fig. 9.2. This Python program builds a PyMC model of the above example, performs Monte Carlo Markov Chain inference, plots the resulting distributions, and saves the result to a PDF file.

```
 1 data = pd.read_csv('./data.csv')
 2 print('loading data.csv')
 3 print(data.head())
 4 print('...')

 6 with pm.Model() as model:
 7     prpro_b0 = pm.Uniform('b0', -2, 2)
 8     prpro_b1 = pm.Uniform('b1', -2, 2)
 9     prpro_sigma = pm.Uniform('sigma', 0.001, 2.0)
10     prpro_y = pm.Normal('y',
11             mu = ((prpro_b1 * data['x']) + prpro_b0),
12             sigma = prpro_sigma, observed = data['y'])
13     inferred_data = pm.sample(return_inferencedata = True)

15 print('creating plots')
16 N = 3 # numbers of parameters (non-data backed variables) in the model
17 fig, ax = plt.subplots(2, N, figsize = (15, 10))
18 ax[0,0].plot(data['x'], data['y'], 'o', label='data', alpha = 0.3)
19 ax[0,0].set_xlabel('x')
20 ax[0,0].set_ylabel('y')
21 ax[0,0].set_title('A scatter plot of the data')

23 az.plot_posterior(inferred_data, var_names = ['b0'],    ax = ax[1,0])
24 az.plot_posterior(inferred_data, var_names = ['b1'],    ax = ax[1,1])
25 az.plot_posterior(inferred_data, var_names = ['sigma'], ax = ax[1,2])
26 fig.tight_layout()

28 print(f'generating a pdf file {__file__}.pdf')
29 plt.savefig(f'{__file__}.pdf')
```

source: prpro.py/reference-example.py

Figure 9.2: *The reference output implementation of code that could have been generated from a* `prpro` *model (abbreviated)*

The first four lines just load the data file. The filename and path are fixed to `./data.csv`. When generating this code, the first four lines need to be emitted independently of the input. Lines 7–12 capture the probabilistic part of the input, in the same order as in the model above. Each expression is transformed from mathematical notation to PyMC syntax. Such translation requires a piecewise traversal of the input syntax tree. Finally, lines 16–17 and lines 23–25 produce the plots. The number of these lines and some values (like `N` in lines 16–17) depend on the number of parameters that we have in the input model. The structure of the lines is almost fixed though, and we could generate them with a for loop, traversing a list of the input parameters.

An example like the one in Fig. 9.2 is called *a reference output implementation*. Before you settle on building a code generator, it is highly recommended to implement one example output manually, and get it to run in the intended context. This helps to solve key interaction problems with the supporting platform and clarifies the architectural division between the static and the dynamic code. Consequently, designing a code generator is much easier. It amounts to identifying the model-dependent parts, and replacing them with the generation logic.

```java
static public String generate (Expression e)
{ return new GeneratorExprSwitch().doSwitch (e); }

static public String generate (Expression e, Declaration context)
{ return stripParens (new GeneratorExprSwitch (context).doSwitch (e)); }

public static String stripParens (String s) {
  if (s.charAt (0) == '(' ) return s.substring(1, s.length () - 1);
  else return s;
}

static class GeneratorExprSwitch extends PrproSwitch<String>
{ ...
  public GeneratorExprSwitch (Declaration d) { this.context = d; }

  private String lhs ()
  { return (context != null) ? context.getName () : ""; }

  private String observed () {
    if (context != null) {
      Model model = (Model) context.eResource().getContents().get(0);
      for (Declaration d: model.getDecls())
        if (d instanceof Data && d.getName().equals(context.getName()))
          return ", observed = data['" + d.getName () + "']";
    }
    return "";
  }
  @Override public String caseCstI (CstI expr)
  { return expr.getValue ().toString (); }

  @Override public String caseCstF (CstF expr)
  { return expr.getValue ().toString (); }

  @Override public String caseVarRef (VarRef expr) {
    String name = expr.getReferencedVar ().getName ();
    if (expr.getReferencedVar () instanceof Data)
      return "data['" + name + "']";
    else return "prpro_" + name;
  }
  @Override public String caseBExpr (BExpr e) {
    String left = generate (e.getLeft ());
    String right = generate (e.getRight ());
    switch (e.getOperator ()) {
      case MINUS: return "(" + left + " - " + right + ")";
      case MULT:  return "(" + left + " * " + right + ")";
      case DIV:   return "(" + left + " / " + right + ")";
      default:    return "(" + left + " + " + right + ")";
  } }

  @Override public String caseNormal (Normal e) {
    String mu = ", mu = " + stripParens (generate (e.getMu ()));
    String sigma = ", sigma = " + stripParens (generate (e.getSigma ()));
    String name = lhs () != "" ? "'" + lhs () + "', " : "";
    return "(pm.Normal(" + name + mu + sigma + observed () + "))";
  }
  ...
```

source: prpro.java/src/main/java/dsldesign/prpro/java/ExprGenerator.java

Figure 9.3: *A visitor translating a prpro expression into a Python expression using the PyMC library. An example implementation in Java*

Exercise 9.1. Print or photocopy Fig. 9.2 and compare it carefully with the input model in the example. Annotate parts of the code that depend on the model, and parts that are fixed. Watch for references to identifiers from the model (clearly model-dependent) and try to speculate which lines depend on the input model entirely (they would not have been there if a line in the input had been missing).

9.2 Code Generation Using Visitors and Recursion

Let us start with discussing how we translate `prpro` expressions into Python. This part of the code depends most strongly on the input. Expressions are defined inductively—bigger expressions are built from smaller ones. The meta-model contains cycles over containment cycles (Fig. 6.3, p. 206). Such structures are best processed by traversals, using visitors or recursion, similarly to type-checking in Chapter 6 and interpretation in Chapter 8.

The idea is to traverse the abstract-syntax tree bottom up, creating a piece of output for each leaf in the tree, and composing them together for internal nodes. Figure 9.3 presents an implementation for `prpro` using the visitor pattern in Java. It assumes that the model is loaded in memory, as an instance conforming to the Ecore meta-model of Fig. 6.3. We are only concerned with translating the expression part of the model here; the core part can be found in lines 28–56. For a simple integer constant expression (l. 28) and for a simple float expression (l. 31), we just translate the value of the literal to a character string. For a variable reference (l. 34) we check whether the variable referred to is a data variable. If so we refer to the data frame `data` in line 37, otherwise we just return the name of the variable, prefixing it with `'prpro_'`. Compare to the corresponding output in Fig. 9.2. In Line 11, parameter `b1` is prefixed, while `x` is turned into a reference to data. For binary expressions (l. 40 and onwards), we first generate the string representation of the left literal, then of the right one, and proceed to compose them with the correct binary operator. We enclose the output in parentheses to avoid problems with operator precedence.

Consider the case of translating a Gaussian density expression in the bottom of the figure. We generate the code for the expression defining the mean first (l. 51). We use a recursive visitor. We proceed similarly for the standard deviation in l. 52. In Line 53, we check whether this normal density is used directly in a top-level let expression. If it is, we generate a string representing the variable name enclosed in quotes. This is needed, because PyMC should receive this name to be able to use it in the library when outputting plots and model graphs (useful for debugging, plots, and summaries). Finally, in Line 54, we compose the entire distribution expression, including a reference to the observed data set, if such exists, using a helper function `observed`. The remaining part of the code provides the helper functions and sets up the generators.

It is instructive to compare this code to the type checker of Chapter 6 (Fig. 6.13, p. 218); especially lines 19–20 in the latter to lines 41–42 here. Both the type-checking and the code generation algorithms perform the same traversal of the syntax tree, but they produce different values. A similar parallel can be drawn to interpreters. We show no interpreter for `prpro`, but see lines 23–24 in the `expr` interpreter (the right column in Fig. 8.7).

In many functional languages the visitor pattern is not an option as overloading and dynamic dispatch are not available. Recursion is a natural alternative. Figure 9.5 shows the corresponding generator in Scala. However, before we discuss it in more depth, let us introduce a complication. After all, we have seen recursive interpreters and type checkers already, and this generator is not going to be very different.

9.3 Memory Management for Code Generation

One common problem with generation of anything but the smallest programs is memory management. A traversal of a model creates small strings in leaves of an AST and recomposes them into larger strings in internal nodes. In a language with automatic memory management (garbage collection) this tends to create a new string object for each composition operator. For instance in Line 54 of Fig. 9.3, five new string objects will be created, not counting the string objects created in lines 51–53. This phenomenon leads to code generators quickly becoming memory inefficient, or even consuming the entire RAM and crashing for large output files. In imperative languages, the standard solution is to use a string builder instead, which uses destructive updates to extend the string in place, instead of producing new values (see Exercise 9.15). In purely functional style, one uses a representation that minimizes polluting memory with intermediate objects.

Think what happens when we concatenate n strings: $s_1 + s_2 + \ldots + s_n$. If the concatenation operator is left-associative, we execute $(\cdots(((s_1 + s_2) + s_3) + s_4)\cdots + s_{n-1}) + s_n$. We create each of the $n-1$ prefix string objects:

$$\begin{array}{l} s_1 + s_2 \\ s_1 + s_2 + s_3 \\ s_1 + s_2 + s_3 + s_4 \\ \vdots \\ s_1 + s_2 + s_3 + s_4 + \cdots \\ s_1 + s_2 + s_3 + s_4 + \cdots + s_n \end{array}$$

The result of the final computation above produces a string which uses memory proportional to the combined size of the input strings. The penultimate row uses memory proportional to the combined size of all but the last one, and so on. The total memory allocated in the process is quadratic in the size of the input strings. These allocations happen too quickly for the garbage collector to free the temporaries efficiently, clogging the system.

What can we do instead? For the strings s_i above, a simple list is itself a perfect representation—we can extend it with new strings in constant time and space. We can output it to a file, string by string, in linear time. Unfortunately, functional lists cannot be extended efficiently at the end, as

```scala
 1 enum StringTree:
 2   case Em
 3   case Lf (s: String)
 4   case Br (pr: StringTree, sf: StringTree)

 6   infix def |+| (sf: StringTree): StringTree = Br (this, sf)
 7   infix def |+| (sf: String): StringTree = Br (this, Lf (sf))

 9   override def toString: String = this.toStringList(Nil).mkString

11   def toStringList (l: List[String]): List[String] = this match
12     case Em => l
13     case Lf (s) => s::l
14     case Br (pr, sf) => pr.toStringList (sf.toStringList (l))

16 extension (pr: String)
17   def |+| (sf: StringTree): StringTree = Br (Lf (pr), sf)
18   def |+| (sf: String): StringTree = Br (Lf (pr), Lf (sf))
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/exprGenerator.scala

Figure 9.4: *An example of an efficient output representation for text files to be used by generators. It avoids trashing the heap with many temporary objects. The `|+|` operator is added for convenience. Lines 6–7 allow suffixing a string tree representation with a string or another tree. Lines 16–18 extend the `String` class, to allow prefixing a string or a string tree with another string*

this also leads to a quadratic cost. During code generation we often need to build up fragments from both ends. We can do this efficiently if we represent the output as a tree, as we already do for the input. Unlike for the input though, the output does not require semantic information. The only thing we need to remember is what is combined with what, so we can use a single abstract meta-model for all output languages. With a tree instance, the generator can proceed efficiently using memory linear in the size of the output. Once it is done, the tree can be serialized to a file in linear time, or converted to a String imperatively (say, using a string builder).

A small and efficient pure tree representation for code generation results is shown in Fig. 9.4. Think of it as if this was a simple meta-model or a language for representing string outputs. We have three kinds of `StringTrees`: a leaf (`Lf`); a branch (`Br`), and empty (`Em`). A leaf stores a single string, a branch combines two trees. We can add a new tree before or after another one, for the same small cost. The empty constructor is practical to provide a value when a part of a generator does not produce any output.

The string tree type implements two operations: concatenation (`|+|`) and serialization to a list (`toStringList`) for easy output to a file. Concatenation of a string or a string tree to a current string tree just creates a suitable branch value (lines 6–7 and 17–18). Only a constant amount of memory is allocated for each new node; the string values are not copied. The function `toStringList` uses a post-order traversal of the tree, to build a list efficiently (prefixing is a cheap operation for functional lists). In Line 9, we add a helper function `toString` that uses an efficient implementation of Scala's `mkString` to create a single string value. This way the entire generator can be both efficient and pure (the impurity is encapsulated in `mkString`).

With this implementation in hand, we create a memory-efficient and pure code generator for `prpro`'s expressions in Scala Fig. 9.5). The generator

```
 1 type DataEnv = Map[String, VectorTy]

 3 def generate (denv: DataEnv, context: Option[Declaration], e: Expression)
 4   : StringTree = e match
 5   case BExpr (left, operator, right) =>
 6     val sLeft = generate (denv, None, left)
 7     val sRight = generate (denv, None, right)
 8     val sOpe = generate (operator)
 9     "(" |+| sLeft |+| sOpe |+| sRight |+| ")"

11   case CstI (n) => Lf (n.toString)
12   case CstF (x) => Lf (x.toString)

14   case VarRef (name) =>
15     if denv.isDefinedAt (name) then
16       "data[’" |+| name |+| "’]"
17     else "prpro_" |+| name

19   case Normal (mu, sigma) =>
20     val sMu = "mu = " |+| generate (denv, None, mu)
21     val sSigma = ", sigma = " |+| generate (denv, None, sigma)
22     val sObserved = observed (denv, context).getOrElse (Em)
23     "(Normal(" |+|varName(context)|+|sMu|+|sSigma|+|sObserved|+| "))"

25   case Uniform (lo, hi) => ...

27 private def generate (ope: Operator): String = ope match
28   case Plus => "+"
29   case Minus => "-"
30   case Mult => "*"
31   case Div => "/"

33 private def varName (context: Option[Declaration]): StringTree =
34   Lf (context.map { decl => s""""${decl.name}", """ }.getOrElse (""))

36 private def observed (denv: DataEnv, context: Option[Declaration])
37   : Option[StringTree] = for
38     decl <- context
39     o = generate(denv, None, VarRef (decl.name))
40         if denv.isDefinedAt (decl.name)
41   yield Lf (s", observed = ${o}")
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/exprGenerator.scala

Figure 9.5: *A recursive implementation of a code generator in Scala translating a* `prpro` *expression into a Python expression that uses the PyMC library. The generator is build on top of the string tree library of Fig. 9.4*

uses |+| instead of + to build the output. Recursion replaces the visitors, resulting in a more concise implementation. The familiar structure remains.

> **Exercise 9.2.** Compare Fig. 9.3 against Fig. 9.5, especially the lines 3–23 in the latter. Compare these to the type checker for `prpro` in Fig. 6.12 on p. 217. Observe how type-checking and code generation perform the same kind of traversal, just creating a different representation (types vs a code string tree).

In the Scala code generator, we chose to use information obtained during type-checking. In Line 15, we consult the data environment `denv` to see whether a data definition exists for a variable. In the Java variant, we chose to extract this information directly from the abstract syntax, arguably

```
def generate (denv: DataEnv, context: Option[Declaration], e: Expression)
  : Doc = e match
  case BExpr (left, operator, right) =>
    val dLeft = generate (denv, None, left)
    val dRight = generate (denv, None, right)
    val dOpe = generate (operator)
    paren (dLeft + dOpe + dRight)

  case CstI (n) => Doc.str (n)
  case CstF (x) => Doc.str (x)

  case VarRef (name) =>
    if denv isDefinedAt name then
       Doc.str ("data[") + quote (name) + Doc.str ("]")
    else Doc.str ("prpro_") + Doc.str (name)

  case Normal (mu, sigma) =>
    val dMu = Doc.str ("mu = ") + generate (denv, None, mu)
    val dSigma = Doc.str (", sigma = ") + generate (denv, None, sigma)
    val dObserved = observed (denv, context)
    paren (Doc.str ("pm.Normal") + paren (
      varName (context) + dMu + dSigma + dObserved))

  case Uniform (lo, hi) => ...

def generate (denv: DataEnv, l: Let): Doc =
  val rhs = generate (denv, Some (l), l.value)
  Doc.str ("prpro_") + Doc.str (l.name) + Doc.str (" = ") + rhs
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/prproGenerator.scala

Figure 9.6: A recursive implementation of a code generator for `prpro` *in Scala using a pretty-printing library* `paiges`*. The helper functions* `varName` *and* `observed` *have been hidden for brevity*

repeating some work already done by the type checker (l. 36, Fig. 9.3). We did what seemed simpler, more concise, in each of these cases. Still, it is common for more involved input languages that the type-checking information influences the code generation. This is particularly so if the input language is untyped while the output language is typed or requires precise memory management (such as C).

The idea of a library of types and combinators to represent generated code can be expanded to a full-fledged system that not only solves the memory representation problems but also allows the output to be structured more easily. One such library is `paiges` for Scala;[1] similar exist for almost any programming language—search online for a "pretty-printing library." Figure 9.6 shows a reimplementation of the generator from Fig. 9.5 using `paiges` instead of string trees. The `StringTree` type has been replaced by the `Doc` type from `paiges`. Otherwise the code is almost identical.

Pretty-printing libraries offer support for automatic line wrapping (hard and soft line breaks), controlling indentation, hangs, and block nesting. They offer convenience operators, so that we do not need to convert all types to strings explicitly. Finally, a number of rendering facilities, including

[1] https://github.com/typelevel/paiges, seen 2022/09

```
 1 def body (denv: DataEnv, m: Model): Doc =
 2   val lets = m.collect { case l: Let => l }
 3   val docs = lets.map { l => generate (denv, l) }
 4   Doc.intercalate (Doc.lineBreak, docs)

 6 def plots (denv: DataEnv, tenv: TypeEnv, m: Model): Doc =
 7   val parameters = tenv.keySet.diff (denv.keySet)
 8   parameters
 9     .toList
10     .zipWithIndex
11     .map { (name, i) =>
12       s"az.plot_posterior(trace, var_names=['${name}'], ax=ax[1,${i}])" }
13     .map { Doc.str _ }
14     .foldRight[Doc] (Doc.empty) (_ / _)

16 def generate (denv: DataEnv, tenv: TypeEnv, m: Model): String =
17   val x = denv.keys.toList (0)
18   val y = denv.keys.toList (1)
19   val N = tenv.keySet.diff (denv.keySet).size
20   s"""print('loading data.csv')
21       |data = pd.read_csv('./data.csv')
22       |print(data.head())
23       |print('...')

25       |with pm.Model() as model:
26       |    ${body(denv, m).hang (4).render (-1)}
27       |    trace = pm.sample(draws = 20000, return_inferencedata = True)

29       |print('creating plots')
30       |N = ${N} # numbers of parameters in the model

32       |fig, ax = plt.subplots(2, N, figsize=(15, 10))
33       |ax[0,0].plot(data['${x}'], data['${y}'], 'o',label='data',alpha=.3)
34       |ax[0,0].set_xlabel('${x}')
35       |ax[0,0].set_ylabel('${y}')
36       |ax[0,0].set_title('A scatter plot of the data')

38       |${plots (denv, tenv, m).render (-1)}
39       |fig.tight_layout()

41       |print(f'generating a pdf file {__file__}.pdf')
42       |plt.savefig(f'{__file__}.pdf')
43   """.stripMargin
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/prproGenerator.scala

Figure 9.7: *The template part of the* `prpro` *code generator. The recursive part, invoked in line 3, is found in Fig. 9.6. The template is slightly abbreviated to fit the figure*

translations to lazy streams or directly to output streams, allow code to be emitted without creation of another in-memory copy of the document.

9.4 Code Generation with Templates

We already know how to translate expressions, creating a small but essential part of our reference example, lines 7–12 in Fig. 9.2. A careful look at the remaining part of that figure reveals that the rest of the code depends less on the input and can be processed without recursive traversals. This situation is common—the output structure is fixed, except for pieces of information selected from the model. While a regular visitor producing code is a program that follows the structure of the input to compose the output out of small pieces, we need the opposite: a program that follows the boilerplate string of the output program, but with some *gaps* populated by computations referring to the input model. We need *templates*.

Interpolated strings as templates. Figure 9.7 shows a template for the top level of the `prpro` code generator. The bulk is found in lines 20–43. We use the simplest template technology—interpolated multi-line strings. A large string in the example looks like the output program. The gaps are marked `${...}`; each contains a Scala expression, such as a call or a variable reference. Most are very simple (lines 33–35). In Line 26, we invoke the recursive code generator of Fig. 9.6. We use `stripMargin`, a helpful method on strings in Scala, which allows us to maintain the indentation of the generator code (as in the file), but produces a different indentation in the output string—the margin left of `|` symbols is stripped from each line in the output.

Exercise 9.3. The generator of Fig. 9.7 assumes two data columns in the data file, and two variables in the model. The variable defined in the last line of the model is the target of the regression. Generalize this to n columns, with $n-1$ regression variables, the last still being the target. Most changes need to be done in the plotting code. Instead of plotting x against y, plot each of the regressors (column variables) against the target (y) separately, changing l. 32 to create $n-1$ plots (adding more plots in the first row of Fig. 9.1). The remaining plots (posteriors) for parameters do not change. Use the function `plots`, lines 6–14, as inspiration for generalizing the first row plots.

Templates are an established technique, particularly in web development. PHP[2] was probably the first famous template language. PHP embeds pieces of executable code in an HTML document. In general, the haps in templates contain meta-code (i.e., code-creating code), which is run at *template instantiation time* to compute the variable parts [4]. In modern GPLs, simple templates can be built by string interpolation and multi-line string literals (C#, Kotlin, Groovy, Scala, Xtend, etc.). Recently, Java 15 has introduced multi-line strings, known there as text-blocks, with support for formatting gaps and automatic adjustment of indentation.

Dedicated template languages. Dedicated template languages offer more functionality than simple string interpolation—besides the ability to evaluate expressions and calls they feature control structures, loops and conditionals, to create regularly structured and optional fragments. Control structures cannot be embedded in interpolated strings. Some template languages integrate directly with Ecore, which allows for very lightweight construction of generators which read a model instance directly (Xpand,[3] Acceleo[4]). Many languages have been created for report generation and web

[2] https://www.php.net/, accessed 2022/09

[3] http://wiki.eclipse.org/Xpand, accessed 2022/09

[4] http://www.eclipse.org/acceleo, accessed 2022/09

```java
class FSMCoffeeMachine {
  static final int INITIAL = 0;
  static final int SELECTION = 1;
  static final int BREWCOFFEE = 2;
  static final int BREWTEA = 3;
  static final int BROKEN = 4;
  static int current;
  static final String[] stateNames = {
    "initial","selection","brewCoffee","brewTea","broken",
  };
  static final String[] availableInputs = {
    "<coin><break>",
    "<tea><coffee><timeout><break>",
    "<done><break>",
    "<done><break>",
    "",
  };
  public static void main (String[] args) {
    @SuppressWarnings(value = { "resource" })
    Scanner scanner = new Scanner(System.in);
    current = INITIAL;
    while (true) {
      System.out.print ("[" + stateNames[current] + "] ");
      System.out.print ("What is the next event? available: " +
        availableInputs[current]);
      System.out.print ("?");
      String input = scanner.nextLine();
      switch (current) {
        case INITIAL:
          switch (input) {
            case "coin":
              System.out.println("machine says:what drink do you want?");
              current = SELECTION;
              break;
            case "break":
              System.out.println("machine says:machine is broken");
              current = BROKEN;
              break;
          }
          break;
        case ...
} } } }
```

Figure 9.8: *Java code generated for the finite-state machine from Fig. 7.2 on p. 238 (see also Fig. 9.17 on p. 353). White space is reduced in the figure and some cases are omitted to conserve space*

programming. Most of these can be effectively used for code generation. Examples include StringTemplate,[5] Velocity,[6] FreeMarker,[7] and JSP.[8] In the Python universe, Jinja[9] is a popular and feature-rich language. Let us use Xtend to create code from `fsm` models. We used Xtend briefly in Chapters 5 and 7. Xtend incorporates Xpand templates as part of its multi-line strings.

[5] https://www.stringtemplate.org/, accessed 2022/09
[6] https://velocity.apache.org, accessed 2022/09
[7] http://freemarker.org, accessed 2022/09
[8] https://jsp.java.net, accessed 2022/09
[9] https://jinja.palletsprojects.com/, accessed 2022/09

```
def static compileToJava(FiniteStateMachine it) {
   var int i = -1
   '''
      import java.util.Scanner;
      class FSM«it.name.toFirstUpper» {
         «FOR state : it.states»
            static final int «state.name.toUpperCase» = «i = i + 1»;
         «ENDFOR»
         static int current;
         static final String[] stateNames = {
            «FOR state : states»"«state.name»",«ENDFOR»
         };
         static final String[] availableInputs = {
            «FOR state : states»
               "«FOR t : state.leavingTransitions»<«t.input»>«ENDFOR»",
            «ENDFOR»
         };

         public static void main (String[] args) {
           @SuppressWarnings(value = { "resource" })
           Scanner scanner = new Scanner(System.in);
           current = «initial.name.toUpperCase»;
           while (true) {
            System.out.print ("[" + stateNames[current] + "] ");
            System.out.print ("What is the next event? available: "
              + availableInputs[current]);
            System.out.print ("?");
            String input = scanner.nextLine();
            switch (current) {
            «FOR state : states»
               case «state.name.toUpperCase»:
                  switch (input) {
                  «FOR t : state.leavingTransitions»
                     case "«t.input»":
                        System.out.println ("machine says:«t.output»");
                        current = «t.target.name.toUpperCase»;
                        break;
                  «ENDFOR»
                  }
                  break;
            «ENDFOR»
            }
           }
         }
      }
   '''
}
```

source: fsm.xtend/src/main/xtend/dsldesign/fsm/xtend/ToJavaCode.xtend

Figure 9.9: *Xtend/Xpand template generating Java code from a finite-state machine instance*

We begin by exploring what code needs to be generated for this example. A reference example is presented in Fig. 9.8. Abstract object-oriented and functional patterns for implementing state machines are readily available, yet we opted to present a low-level scheme that could also be realized in a low-level language. This code can easily be translated to C (Exercise 9.8). In the figure, lines 2–6 define symbolic names for integers used to represent the current state (in C it would have been natural to use preprocessor symbols). The current state is stored in the variable `current` declared in Line 7. Lines 8–10 define names of states as strings (so that we can print them), and lines 11–17 list available actions for each state (also for printing messages). The behavior of a state machine is realized in lines 18–42. A machine is executed by a non-terminating loop (l. 22). In each iteration, we print the current state and the available actions (l. 23–26), and wait for the user to input the next one (l. 27). We decide what to do next, depending on what is the current state, using a switch. In the initial state, we react to `coin` and `break`. Each reaction prints a message and changes the current state. If an action provided by the user is not handled in the model, we just repeat the iteration, asking the user for another action.

Figure 9.9 shows a template, which generates code like that in Fig. 9.8. The three for-loops at the top generate the list of state identifiers, the list of state names, and the list of available actions. Xpand control keywords in the template, the gaps, are enclosed in French quotes, *guillemets*, « and ». These are rarely used in programming languages, so the chance of a clash with the generated language is minimized. Furthermore, note that without for-loops, with usual interpolated strings, we would have to invoke a helper function for each of these blocks, making it much harder to appreciate the structure of a produced file. Xpand also handles indentation automatically. The indentation of the entire template string (visible in l. 3) will not influence the indentation of the output (so the class declaration produced in l. 5 will not be indented in the output file, as shown in Fig. 9.8). Most of the loop code is static, independent of the input model, but for each possible current state we generate a case block in the switch statement, extracting the necessary information from the model as we go.

9.5 Case Study: Robot

Let us combine the techniques discussed above to build a larger case study: a code generator for the `robot` language. Recall that we have already built an interpreter for `robot` in Chapter 8. In Sect. 8.1, we have presented a platform implementation providing the basic primitives for controlling a TurtleBot. We have used this implementation (Fig. 8.5, p. 299) together with ROS as the basis for the interpreter. Our adaptation layer supports initialization of the model and the robot, maintaining the controller state, logging, sending commands to the robot, deciding how long they should be active, and reading sensors. We will now build a Python code generator, using the same platform to support the generated code.

A reference example implementation for `robot`. The same way as Fig. 9.2 shows a reference example for `prpro`, Fig. 9.10 shows a reference example for `robot`, using the random walk model of Fig. 2.2 (p. 30) as the input. Let us understand what the key parts are. In the example, we use integers to identify modes (lines 1–4). This is not strictly necessary in Python, but would be natural, for instance in C, if we wanted to create a compact and efficient

```python
_RANDOM_WALK = 1
_MOVING_FORWARD = 1*2
_AVOID = 1*3
_SHUT_DOWN = 1*5
class RandomWalk(TurtleBotPlatform):
    ...
    def execute_AVOID(self):
        if self.mode_pc == 0:
            self.info('AVOID[0]')
            direction = -1.0
            velocity = 0.1 * direction
            self.engage(velocity, float(2))
            self.mode_pc = self.mode_pc + 1
            return True
        if self.mode_pc == 1:
            self.info('AVOID[1]')
            self.random_rotation()
            self.mode_pc = self.mode_pc + 1
            return True
        return False

    def run(self):
        while True:
            self.executor.spin_once(0.0)
            if self.tm_latch in self.timers:
                self.executor.spin_once(1.0/self.FREQ)
                continue

            # actions and sub-mode activations
            if self.mode==_RANDOM_WALK and self.execute_RANDOM_WALK():
                continue
            if self.mode==_MOVING_FORWARD \
                    and self.execute_MOVING_FORWARD():
                continue
            if self.mode == _AVOID and self.execute_AVOID():
                continue
            ...
            # reactions
            if self.active(_MOVING_FORWARD):
                if self.ev_obstacle:
                    self.info('Reacting to an obstacle!')
                    self.ev_obstacle = False
                    self.activate(_AVOID)
                    continue

            # continuations
            if self.mode == _AVOID:
                self.activate(_MOVING_FORWARD)
                continue
```

source: robot.turtlebot3/dsldesign_robot_turtlebot3/controller.py

Figure 9.10: *The core part of the reference controller implementation of the random walk model of Fig. 2.2, p. 30*

implementation. Instead of using consecutive integers though, we use consecutive prime numbers. The identifier of each mode is chosen to be its own prime number multiplied by the identifiers of its container modes (parents). This way, checking whether one mode is a parent of another amounts to

```
class Controller(TurtleBotPlatform):
    ...
    {% for m in ctx.modes %}
    def execute{{m.SNAKE_NAME}}(self):
        {% for a in m.actions %}
        if self.mode_pc == {{loop.index0}}:
            self.info('{{m.name}}[{{loop.index0}}]')
            {% filter indent(8) %}
            {{generate_action(a)}}
            {% endfilter %}
            self.mode_pc = self.mode_pc + 1
            return True
        {% endfor %}
        {% for sm in m.modes if sm.initial %}
        self.info('{{m.name}}[initializing submode {{sm.name}}]')
        self.activate({{sm.SNAKE_NAME}})
        return True
        {% else %}
        return False
        {% endfor %}
    {% endfor %}

    def run(self):
        while True:
            self.executor.spin_once(0.0)
            if self.tm_latch in self.timers:
                self.executor.spin_once(1.0/self.FREQ)
                continue

            # actions and sub-mode activations
            {% for m in ctx.modes %}
            if self.mode=={{m.SNAKE_NAME}} and \
                    self.execute{{m.SNAKE_NAME}}(): continue
            {% endfor %}

            # reactions
            {% for m in ctx.modes if m.reactions %}
            if self.active({{m.SNAKE_NAME}}):
                {% for r in m.reactions %}
                {% if r.trigger == ctx.Robot.Event.EV_CLAP %}
                  ...
                {% elif r.trigger == ctx.Robot.Event.EV_OBSTACLE %}
                if self.ev_obstacle:
                    self.info('Reacting to an obstacle!')
                    self.ev_obstacle = False
                {% endif %}
                    self.activate({{r.target.SNAKE_NAME}})
                    continue
                {% endfor %}
            {% endfor %}

            # continuations
            {% for m in ctx.modes if m.continuation %}
            if self.mode == {{m.SNAKE_NAME}}:
                self.activate({{m.continuation.SNAKE_NAME}})
                continue
```

source: robot.py/controller.py.jinja

Figure 9.11: *The core part of the code generation template for* `robot`*, mostly corresponding to the parts of the reference implementation in Fig. 9.10*

checking whether one identifier is divisible by another. This check is encapsulated in the function `self.active` (not shown but used in a few places).

For each mode, an execution function enforces the encapsulated behavior. Recall the obstacle avoidance mode:

```
Avoid {
  move backward for 1 s
  turn by random (-180,180)
} -> MovingForward
```

In `Avoid`, two actions are executed. First the robot moves backwards for 1s, then it rotates through a random angle. Subsequently, the control switches to the `MovingForward` mode. The corresponding code is found in lines 7–20. The variable `self.mode_pc` records the action being executed. This is often called a program counter, thus `pc`. If we are at the first action (`mode_pc==0`, Line 8), we log this fact, set the direction of motion backwards, calculate velocity, and call the platform's `engage` to move the robot. After executing the line, we increment the counter and exit (l. 13-14) to allow time to pass. The controller will resume the execution after the set delay. Then the second action is executed: we perform a random rotation (l. 15–19). For simplicity, we ignore the action's parameter, the range of random angles for rotation. We just invoke `self.random_rotation` from the platform to perform an arbitrary turn. It is an interesting exercise to first evaluate the argument expression, and then use it to control the size of the rotation in `random_rotation`.

The main execution loop is found in the `run` function. It has four parts: delaying until actions are complete (l. 24–27), executing code inside modes upon activation (29–36), reacting to external events (38–44), and switching to a continuation mode, if there is one (46–49). We discuss them in order.

In every iteration, lines 24–27 release control to allow callbacks to be activated. This is necessary because we use a sequential non-preemptive scheduler. If any actions are active (l. 25), we wait for their completion, ignoring any other events; sleeping a short cycle to avoid busy waiting and restarting the loop. This is a bit convoluted, and it is meant to be! An implementation of a DSL will always be dense with domain-specific logic and behavior from your problem domain. Fortunately, we do not need to understand this to understand the example. From the code generation perspective, l. 24–27 are static code that just needs to be output.

Once no actions await completion, we check which mode is active (lines 30–35) and execute its body. An execute function (like the one in l. 7–20) returns true if more actions remain to be executed, and false when the body execution is completed. After a line of model code is executed, we restart the interpreter loop (`continue`) to check whether no actions await completion. If the check fails, we continue executing the other active modes. This fragment of code is sorted topologically by modes, so the top-level modes will be executed before nested modes.

In lines 39–44, we check what mode is active, and whether any external events have been registered that are required in this mode. Recall the

`MovingForward` mode in the model. It contains a reaction rule that changes the mode to `Avoid` whenever an obstacle is detected:

```
-> MovingForward  {
    move forward at speed 10
    on obstacle -> Avoid
}
```

The variable `selv.ev_obstacle`, tested in l. 40, is set when the platform object detects an obstacle. This happens in a process parallel to the controller. The implementation for the rule of `MovingForward` (l. 39–44) produces a log entry, resets the event variable, and activates the target model `Avoid`.

Finally, in l. 46–49, we check whether the active mode has a continuation. If we got to this point, it means that the actions of the mode have been completed. If a continuation is found, we activate it.

> **Exercise 9.4.** Print or photocopy Fig. 9.10 and mark parts that depend on the model, and parts that are fixed. Watch for references to identifiers from the model (clearly model-dependent) and try to speculate which lines depend on the input model entirely (they would not have been there if a line in the input had been missing).

The `robot` *code generator.* We use Jinja to implement the code generator for `robot`. Jinja is a dedicated template language for M2T transformations implemented in Python. Jinja supports inheritance of templates. Inheritance allows definition of high-level structure and its refinement for various outputs—this helps to modularize code generators. Jinja is an external DSL—Jinja code is not Python code, but Python code can be invoked from the template's gaps. The engine supports template comments (not rendered in the output), white-space and indentation control (so that both the input and the output code is readable), and the usual set of control structures (loops, conditionals, and compile-time macros).

Figure 9.11 shows a template for `robot`, which for the random walk model would generate the code in Fig. 9.10. Before understanding it in detail, try to relate the main parts to the reference example. There is a part generating the execute methods (l. 4–21), a part generating the header of the while-loop (24–28), a part generating execution of mode actions (30–34), a part generating the reaction handling (36–50), and a part generating the continuation code (52–56).

As Jinja is a template language, the top-level code is not executed. It is the template's static output, emitted during the instantiation. The executable code is enclosed in braces and percentage signs, `{% ... %}`. Consequently, the first code executed during instantiation is found in l. 3. It opens a for-loop similar to what we have seen in Xtend/Xpand. (The ellipsis . . . is part of neither the Jinja language nor its output—it indicates that the template is abbreviated in the figure.) The loop iterates over the modes listed in `ctx.mode`, made available by the caller. The `ctx` object comes from our library and provides the logic and data needed during generation. See below for more details. The for-loop effectively creates an `execute_XXX` method for each mode found in the model.

```
{% macro generate_action(action) %}
{% if  ctx.isMove(action) -%}
    direction = {{1.0 if action.forward == True else -1.0}}
    {% if action.speed %}
    model_speed = {{ctx.generate_expr(action.speed)}}
    velocity = model_speed * 0.008 * direction
    {% else %}
    velocity = 0.1 * direction
    {% endif -%}
    {% if action.duration %}
    self.engage(velocity, float({{ctx.generate_expr(action.duration)}}))
    {%- else %}
    self.engage(velocity)
    {%- endif %}
{%- elif ctx.isTurn(action) -%}
    self.random_rotation()
{%- elif ctx.isDock(action) -%}
    self.return_to_base()
{%- endif -%}
{%- endmacro -%}
```

source: robot.py/controller.py.jinja

Figure 9.12: *The Jinja macro compiling a single action execution in* `robot`

A nested loop iterates over lines of action code in each mode (l. 6–13). Lines 14–20 initialize a nested mode (not shown in our reference example) and return. In several places, we see another kind of gap, enclosed in double braces, `{{ ... }}`. These gaps contain expressions that evaluate to strings subsequently incorporated into the output of the template. For example in l. 4, we access a capitalized name of the mode to create the function name; `MovingForward` becomes `_MOVING_FORWARD` and we produce `def execute_MOVING_FORWARD`. The values in `m.SNAKE_NAME` are precomputed in our context object for each mode, before the generation starts.

Read the rest of the template and relate it to the reference implementation in Fig. 9.10 to see how the latter emerges at instantiation time. Let us just discuss the new constructs here. In Line 8, a filter adjusts indentation (many filters are supported in Jinja). Line 9 invokes a macro, a sub-template, `generate_action`. Its definition is shown in Fig. 9.12. In Line 18, we use an else-clause for a for-loop. This construct, not usually seen in GPLs, is useful in template DSLs to emit some text when a loop has not been entered. This particular loop includes a filter in l. 14, which might skip all modes. In code generation, we often want to output some default text if a collection we iterate over is empty. In lines 40–46, we use a sequence of if, else-if, and else branches, to select a piece of code conditionally. Concretely, we select the reactions to events that pertain to a given mode in the model.

The template is supported by a small runtime implemented manually (Fig. 9.13). The runtime provides support functions that load the model from a file and pre-compute values needed for generation. This includes deriving a flat list of modes (`def __modes`), generating prime numbers (`__modes`) and using them to calculate identifiers for modes (`__base_ids`, `__derive_ids`). At this time we also calculate capitalized state names (l. 23). Functions `isXXX` detect the type of an action. They are used to decide

```python
class GeneratorCtx:
    def __init__(self, fname):
    ...  self.__modes(); self.__base_ids(); self.__derive_ids(self.model)

    def primes(self): ...
    def SNAKE(self, name):
        name = re.sub(’(.)([A-Z][a-z]+)’, r’\1_\2’, name)
        name = re.sub(’([a-z0-9])([A-Z])’, r’\1_\2’, name)
        return "_" + name.upper()

    def __modes(self):
        self.modes = []
        old_modes = [self.model]
        while old_modes:
            children = [ parent.modes for parent in old_modes ]
            new_modes = [ m for modes in children for m in modes ]
            self.modes.extend (old_modes)
            old_modes = new_modes
    def __base_ids(self):
        prime = self.primes()
        for m in self.modes:
            m.state_id = next(prime)
            m.SNAKE_NAME = self.SNAKE(m.name)
    def __derive_ids(self, parent):
        for m in parent.modes:
            m.state_id = f"{parent.state_id}*{m.state_id}"
            self.__derive_ids(m)

    def isMove(self, a): return isinstance(a, self.Robot.AcMove)
    def isTurn(self, a): return isinstance(a, self.Robot.AcTurn)
    def isDock(self, a): return isinstance(a, self.Robot.AcDock)

    def generate_expr(self, expr):
        if isinstance(expr, self.Robot.RndI):
            return "(random.randrange(0, 2000) / 1000.0)"
        elif isinstance(expr, self.Robot.CstI): return str(expr.value)
        elif isinstance(expr, self.Robot.Minus):
            return - self.evaluate_expr (expr.aexpr)
        elif isinstance(expr, self.Robot.BinExpr):
            left = str(self.evaluate_expr (expr.left))
            right = str(self.evaluate_expr (expr.right))
            if expr.ope == ’+’: return ’(’ + left + right + ’)’
            elif expr.ope == ’-’: return ’(’ + left - right + ’)’
            ...
    def generate_trigger(self, ev):
        if ev == self.Robot.Event.EV_CLAP: return "ev_clap"
        if ev == self.Robot.Event.EV_OBSTACLE: return "ev_obstacle"

if __name__ == ’__main__’:
    env = Environment(loader = FileSystemLoader("."),
            trim_blocks = True, lstrip_blocks = True)
    template = env.get_template("controller.py.jinja")
    print(template.render(ctx = GeneratorCtx("random-walk.xmi")))
```

source: robot.py/generator.py

Figure 9.13: *A runtime for the* `robot` *code generator. It provides support functions, from the top: initialization (loading the model and pre-computing values needed for generation), generating prime numbers, deriving a flat list of modes from a tree, calculating identifiers for modes, detecting which type a particular action has, generating code for expressions and for triggers. Finally, it loads and instantiates the template (bottom)*

The Main Benefits of Code Generation

- *No dependency on the source language:* When we generate code, the system using it no longer depends on the input DSL. This might be desirable to protect intellectual property (input models), or just to keep the target infrastructure simple. When using an interpreter the target system still needs to contain the entire DSL implementation, including the meta-models, front-end stages, and the models themselves.
- *Performance:* Generated code is faster than interpreted code, and when optimization algorithms are used during code generation, it may be made fast as, or faster than, human-written code.
- *Code volume*: If needed, generated code can be aggressively optimized for size. The generated system only has to contain the parts relevant for the given input model. An interpreter normally supports all the features of the input language, even if the model of a given system does not use them all.
- *Overcoming restrictions of the target language:* If the target language is not Turing-complete we can benefit from implementing a code generator in a more expressive language. Whatever elements could not be computed in the target language we can pre-compute at compile-time. For instance, if the target language is the DIMACS format of a SAT-solver, it is impossible to implement an interpreter in it, but we can implement a code generator and use a suitable solver.

source: Stahl and Völter [11]

what code to generate in Fig. 9.12. Lines 33–47 implement a recursive generator of expression code, very similar to the recursive generators shown above for `prpro`.[10] Finally, in lines 49–53, we show the code for loading a template and instantiating it (`render`). For most of the time, this is just regular Python code that perform calculations and helps to keep the gaps in the template small, so that the template code remains readable.

Exercise 9.5. The generator in Fig. 9.11 always creates an `execute_XXX` method for a mode, even if the mode has no actions to be executed. This leads to creating empty methods returning `False` and making the check in lines 32–33 needless—it always fails. Optimize the code generator to not emit the empty execute methods and to avoid the check in l. 32-33 if a mode contains no actions to execute.

9.6 Guidelines for Implementing DSL Code Generators

Most of the guidelines presented in Sect. 8.5 apply to code generators, too. Below we add a bit more advice specific to code generators.

A code generator or an interpreter? It is typically cheaper to understand, design, and maintain an interpreter than a code generator. A key advantage of interpreters is the single-stage execution. A generator implements the dynamic semantics in two stages: the actual generation first, followed by the execution of the generated code second. On the other hand, code generators allow smaller and more efficient implementations of the model. The benefits of code generation are summarized in the box on p. 341. If in doubt, we recommend implementing an interpreter first. ***Guideline 9.1***

[10] This generator may *almost* be turned into an interpreter executed at compile time and producing a constant value, if not the random number generator that has to be executed at runtime.

Guideline 9.2 *Implement a reference example.* Guideline 8.1 on p. 311 states that a reference example implementation is a good way to architect the platform API, and to divide the dynamic and static parts of the semantics implementation. For code generators, an additional benefit is that the reference implementation presents an expected output. It is much easier to create a single example output than to build a generator. Yet, despite the simplicity, the reference example advances you substantially towards an actual generator.

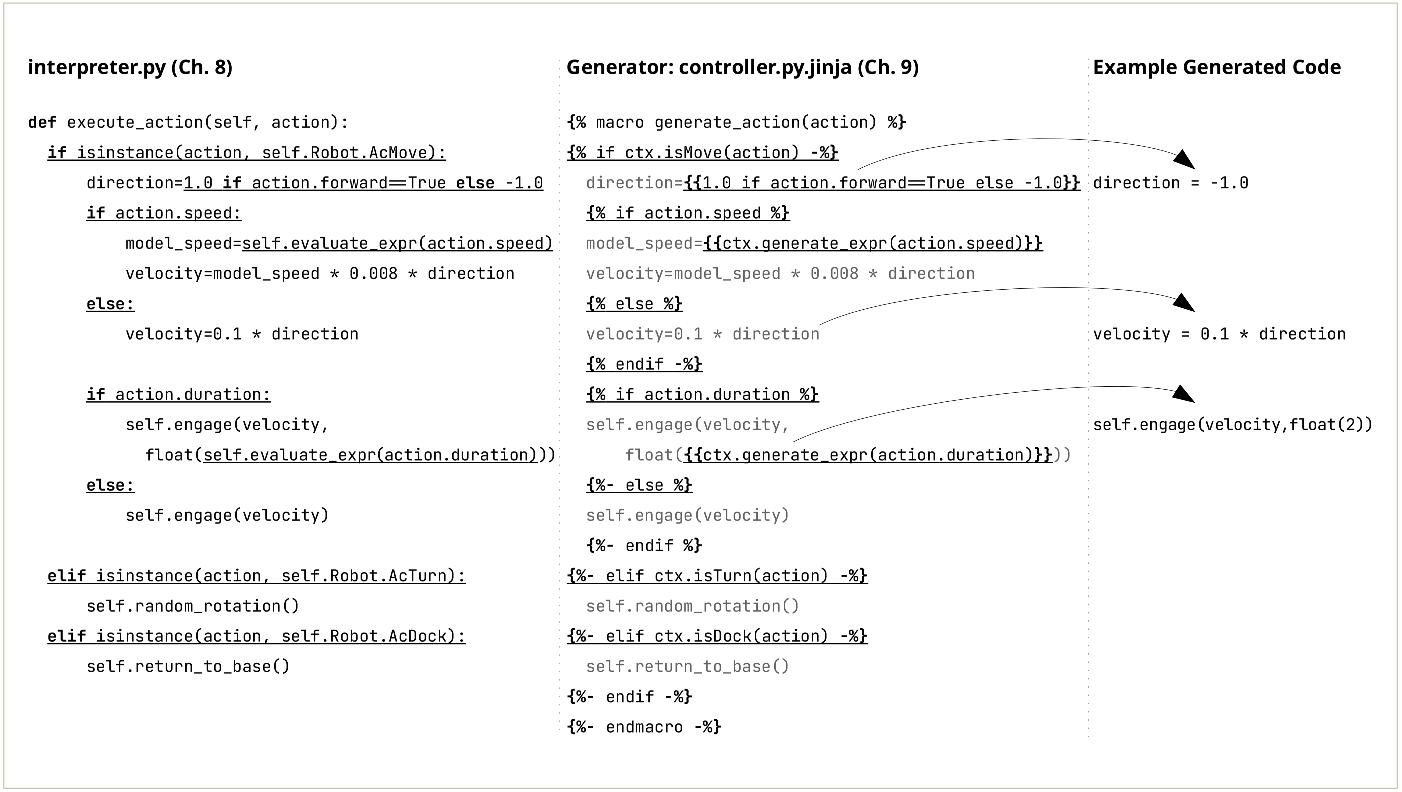
When creating a reference example for generation, maintain a mindset *as if the generator existed*. Be mechanical, not smart, in all your coding decisions. If any smartness is needed to decide what should be generated, mark it as a needed addition to the runtime library (as we did for generating prime numbers in the `robot` case study, Fig. 9.13).

Guideline 9.3 *Even if a code generator is needed, building an interpreter helps to build a better code generator faster.* If the generator is complex and overwhelming, consider implementing an interpreter first. If possible, use the same language as the target of the generator. An interpreter is often simpler to build. It will (i) give you a clean implementation of the execution rules to translate to the generator, (ii) allow you to separate the runtime platform from the dynamic part, (iii) provide parts that can be reused in the generator (types, notions of state, even some evaluators), and (iv) serve as a test oracle for the generated code. Do not build any interpreters though, if it is straightforward to create a generator.

Guideline 9.4 *Use the reference example and the interpreter to understand variability and binding times.* Once you have a reference example, perform a simple *binding-time analysis* [8]. A binding-time analysis gives a systematic way to derive a code generator. It has two steps (which can be used independently, if you only have an interpreter, or only have a reference example).

The first step uses the reference example to understand the structure of the template. Mark the parts of this code that depend on the input model, to distinguish them from the parts that are static. Then consider what information from the model should enter each gap. In the second step, consult the interpreter to see how to fill the template gaps that require complex logic. Finally, turn the reference example program into a template, by replacing the marked parts by generator expressions referring to the input program, and converting parts of the interpreter into compile-time macros.

We have applied this procedure to all examples in this chapter. This is why you have been asked to solve exercises marking static and variable parts of the `prpro` and `robot` reference examples. These exercises help you to find elements that are not static. Let us now investigate how to exploit an interpreter, using Fig. 9.14, to find how to generate complex code. The left column shows an action executor from the interpreter of `robot`. The middle column shows the corresponding template in the `robot` code generator. The right column shows the code generated for the random walk example, for action `AcMove`. In the interpreter code (left) we underline the expressions that *can* be evaluated at generation time for a particular model. We leave the



interpreter.py (Ch. 8)

```
def execute_action(self, action):
  if isinstance(action, self.Robot.AcMove):
      direction=1.0 if action.forward==True else -1.0
      if action.speed:
          model_speed=self.evaluate_expr(action.speed)
          velocity=model_speed * 0.008 * direction
      else:
          velocity=0.1 * direction

      if action.duration:
          self.engage(velocity,
            float(self.evaluate_expr(action.duration)))
      else:
          self.engage(velocity)

  elif isinstance(action, self.Robot.AcTurn):
      self.random_rotation()
  elif isinstance(action, self.Robot.AcDock):
      self.return_to_base()
```

Generator: controller.py.jinja (Ch. 9)

```
{% macro generate_action(action) %}
{% if ctx.isMove(action) -%}
  direction={{1.0 if action.forward==True else -1.0}}
  {% if action.speed %}
  model_speed={{ctx.generate_expr(action.speed)}}
  velocity=model_speed * 0.008 * direction
  {% else %}
  velocity=0.1 * direction
  {% endif -%}
  {% if action.duration %}
  self.engage(velocity,
      float({{ctx.generate_expr(action.duration)}}))
  {%- else %}
  self.engage(velocity)
  {%- endif %}
{%- elif ctx.isTurn(action) -%}
  self.random_rotation()
{%- elif ctx.isDock(action) -%}
  self.return_to_base()
{%- endif -%}
{%- endmacro -%}
```

Example Generated Code

```
direction = -1.0


velocity = 0.1 * direction


self.engage(velocity,float(2))
```

Figure 9.14: A binding-time analysis for the `robot` action execution. Left: The interpreter code executing actions. We mark parts that can be entirely evaluated once the model is known. Center: Convert the marked parts to generator macros (here in Jinja). Right: The result of rendering the template macro for a move action. The code generator takes the same decisions statically as the interpreter does dynamically

code that does not refer to the model unmarked. Each underlined fragment is then converted into a generator macro in the middle column. Now, the processing of an action happens in two stages: the generator evaluates whatever it can and emits the rest (the right column).

Guideline 9.5 *Choose templates where the output structure is independent of the input, traversals when the input structure dominates the output.* The control flow during a traversal of an input AST is driven by the structure of the input model. The control flow during template instantiation follows the order of elements in the template. Logically then, whenever the structure of the output is fixed, and just needs a few gaps to be populated, a template is a suitable governing solution for a generator. However, for intricate structures (especially expressions, but also other inductively defined structures), the generated code has almost no static part, but is entirely driven by the input. For these fragments of the input language, use recursion or visitors.

Guideline 9.6 *Generate as little code as possible, integrate the rest.* In most model-driven architectures, a relatively small part of the system depends on the input model directly. It is recommended to generate only the variable parts of the system. The static parts should be coded in a conventional manner, once and for all. Design as much as possible of the output manually as a runtime library (the platform). Keep it in separate files and directories, and integrate it with the generated code using mechanisms available in the target language (calls, callbacks, class extensions, inheritance, generics, etc.). The larger the static part of your system, the easier it is to test and maintain it.

Guideline 9.7 *Generate code readable for programmers, with clear abstractions, correct indentation, and comments.* Engineers are skeptical of magic. Programmers can understand readable code and verify that it does what it should. This raises trust and confidence in your generator. Understanding helps you and your users to debug the code if anything goes wrong [2]. It is unrealistic to assume that developers will never look at the generated code.

Earlier in the chapter, we ensured that generators produce correctly indented commented code using suitable abstractions. In Fig. 9.3, we use a helper `stripParens` to remove unnecessary outermost parentheses to decrease clutter. We generate names for magic integer constants, and keep comments. These comments, names of states, and names of outputs help to trace the output to the input model elements. For more complex DSLs, you should generate comments or annotations in the output code that link to the input model elements unambiguously, say using URIs, for justification and debugging purposes. Such links are called *traceability links* (see Chapter 7).

It is important though not to compromise readability of the code of the generator itself. A good M2T language, like Jinja, allows white space to be adjusted so that both the input and the output code remain readable.

Exercise 9.6. Our generators for `prpro` (Fig. 9.6, Fig. 9.3) produced too many parentheses for expressions. Let us fix this. Map the operators in `prpro` to integers representing precedence. For instance, addition could be mapped to 1, multiplication to 2. Pass the precedence of the surrounding expression to the generator, and only add the parentheses if the precedence of the context is higher than that of the expression created. This way you will only add needed parentheses. This is best implemented by modifying the generator of Fig. 9.6.

Guideline 9.8 *Remember security, especially code injection attacks.* Code generators are susceptible to code injection attacks, since parts of the output depend on the input model. An action name in `fsm` could be crafted by a malicious attacker to execute arbitrary code on the machine running the generated code. Always consider whether the input models are coming from a trusted source, and whether the generated code is running on a vulnerable machine. As a default, assume that any machine is vulnerable. Use escaping support in the template language or in another established library to sanitize the strings incoming from the model. It is best to rely on existing trustworthy sanitizers, as getting a sanitizer to be watertight is known to be very difficult.

Another line of defence is to forbid dangerous inputs during input validation (parsing and static constraints). Watch for character strings and special symbols that can appear in the input model, especially in literals and names. Confront them with the symbols and keywords of the target language. Do they really need to be allowed in the input? Pay special attention to symbols changing context in the input strings (comments, quotes, other punctuation).

Exercise 9.7. Construct an `fsm` model in the xmi representation such that code generated from it compiles, but throws a runtime exception. Discuss eliminating this attack at the time of parsing, constraint checking, and code generation.

Guideline 9.9 *Automate the build process fully, including code generation.* When integrating a code generator into the build process of the target application, automate the entire process, including invoking the generator and compiling the output. Avoiding manual steps in the workflow decreases chances that people will stop using the generator over time, resorting to manual modifications of the generated code. Requiring manual steps makes it harder for people to experiment with different models, defeating the main purpose of using a DSL and model-driven engineering [11].

Guideline 9.10 *Do not use protected code blocks. Integrate the generated code with static code using other methods.* Some code generators (including the built-in generator of Ecore) support so-called *protected output blocks*. These are marked fragments in the generated code that can be modified or completed after the generation. The generator guarantees that its subsequent runs will not overwrite these changes. This seemingly attractive facility has many pitfalls [11]. First, it blurs the boundary between the generated and static code for future maintainers of your system. Second, it requires that the generated code is version controlled. Third, it leads to code loss eventually, when model elements change in the input so that the part of the model to which your protected code is linked disappears.

We cannot recommend protected code blocks. Instead, integrate generated code with your customizations using the linker of the target language.

Use calls from the generated code to your code or vice versa, and other available mechanisms of the target (extension methods, C++-style templates or generics, mixins, partial type declarations, sub-typing, interfaces, design patterns like facade, adapter, decorator, factory, control inversion, etc.).

Guideline 9.11 *Avoid complex input manipulation in the generator.* If you need to perform complex computations in the generator then your input is likely not well prepared. Rather adapt the input before the code generation, by running a suitable M2M transformation (Chapter 7) that translates the input into an intermediate representation that is semantically closer to the output—see Guideline 8.2 on p. 311. If the alignment of the output and input is good, but complex logic is required because of non-trivial semantics, place this in the code generation support library as we did in Fig. 9.13 for `robot`, keeping the template itself as clean and readable as possible.

9.7 Quality Assurance and Testing for Code Generators

Testing a domain implementation is easier than testing an interpreter, which is easier than testing a code generator (Chapter 8). A static domain implementation is often just a library, which is amenable to standard testing methods. The added difficulty of testing an interpreter is caused by its correctness depending on the input model. The added difficulty of testing generators stems from the two-stage execution process: bugs manifest themselves in the generated code, but the causes of bugs (faults) are in the generator. However, running a debugger on the generator is most often unhelpful. It can explain how the code is created, but not why the created code is wrong. The debugging process for code generators starts in the generated code, where the reasons for failure need to be identified, and then continues into the generator, where the root cause has to be understood and fixed. Since the generator code lacks even basic IDE feedback for the target language (syntax-checking, type-checking, etc.), errors can go unnoticed for a long time.

It is challenging to construct test oracles for code generators. A code generator is correct if the created program is correct—a property hard to formalize and automate. The most basic strategy, often used in compiler implementation, is then to avoid writing clever oracles altogether, but to store the baseline result and to compare against it textually (*comparison with baseline*). In this strategy, the generated code needs to be manually reviewed and stored for comparison as a baseline. Then these baseline samples are compared to the output of the code generator for the corresponding inputs, failing if the output differs from the baseline. This strategy works only for very simple languages, or for mature and stable code generators, where changes are minor and rare. It is not really a testing strategy, but a change-monitoring strategy. Tests fail when the output changes, even if not in a buggy way. Then the human has to verify outputs that change (the actual manual test) and update them again in the baseline collection.

We need more systematic methods than *comparison with baseline* to test generators. Let us stratify the testing process for generators into three stages, of increasing precision and difficulty (mostly inspired by Ratiu, Völter, and Pavletic [10]): (i) robustness of the code generator, (ii) structural correctness of the output, and (iii) structural correctness of the generated code.

Robustness of code generators. The lowest level of ambition when testing code generators is to ensure that the generator does not crash.

Definition 9.1 (Robustness). *A code generator is* robust *if for each input model free of validation (static) errors the generator does not throw any exceptions (or otherwise crash) during execution.*

To test robustness we can feed the generator with diverse input models and monitor safe termination. Random model generation or synthesis helps here, but one needs to take care to create models that are correct inputs (Chapter 5). Ratiu, Völter, and Pavletic [10] report from a case study that it took only a few hundred randomly generated models before they were able to identify about a dozen robustness issues. While robustness issues may not disclose the most intricate problems in a generator, they can be detected quickly and automatically, so defending against them is a natural place to start.

Structural correctness of the generated code. Our second line of defense is checking whether the generator produces a reasonable output—not a program that makes sense from the domain perspective, but a program that looks acceptable for the target language infrastructure.

Definition 9.2 (Structural Correctness). *An automatic code generator maintains structural correctness of the output language if the generated code conforms to the static semantics requirements of the target language: it parses, satisfies static constraints, and type-checks.*

To test structural correctness when the target is a compiled GPL, check whether the generated output compiles for many input models. It helps if your target language is equipped with an expressive type system, and if the platform and the generated code use this type system to capture semantic correctness of the code as much as possible. Then the compiler will detect intricate problems for you, even without running the generated code! In this sense, C++ is a better target language than C (if you use the modern type-system features of C++), and Rust is a better target language than C++. Similarly, Scala can be a more interesting target than Java, and TypeScript than JavaScript. Dynamically typed languages are less useful when it comes to checking structural correctness (although for Python there exist several static checkers independent of the compiler). The situation is similar if another DSL is the target: we run its tool chain to establish whether the model parses, type-checks, and satisfies any static semantics constraints.

Testing structural correctness of the generated code can catch malformed output, and occasionally a semantic problem thanks to types. However, static semantics constraints are often weak. Being problem-independent, they are unlikely to catch any mistakes in your understanding of the domain or the requirements. For this, we need to directly test the dynamic semantics.

Semantic correctness of the generated code. Ultimately, there is no way to avoid testing the semantic correctness of a code generator. Semantics is the most idiosyncratic aspect of our DSL, but also the one that captures the domain properties and user requirements.

Definition 9.3 (Semantic Corrrectness)**.** *A code generator is semantically correct if it always produces programs (models) that correctly capture the meaning of the input domain-specific programs (models).*

Consider four increasingly costly strategies for testing semantic correctness:

- *Create dedicated tests for each property:* For each requirement create an input model (a test case) exhibiting the requirement and write a test *for the output program* to check whether it exhibits the desired property. Executing this test requires running the code generator.
- *Generate assertions in the output programs:* If the target language is expressive enough to support assertions, consider capturing your assumptions and requirements in the assertions in the generated code. These need to be added *to the generator*. This way you can test the output program by running it and checking whether the assertions pass.
- *Support assertions in the input models:* Input-specific requirements *for the output* can be formalized in the DSL models if the DSL supports assertions (alternatively an accompanying test DSL can be created). A code generator can then transfer these assertions to the output, increasing the power of the previous technique. But remember that the assertions in the input open a surface for code injection attacks. Deactivate them in the generation of production code.
- *Property-based testing using abstract-syntax representations*: This is the most expensive method and only pays off for high-assurance domains and DSLs. Use an M2M, not an M2T, transformation. An M2M transformation produces instances, which can be inspected using constraints. Write properties relating the input and the output AST, and use property-based testing with random model generation to test them. For instance, you can verify *traceability*: that for each input model element the corresponding output model element exists, and that each variable output element has a corresponding input element justifying its existence.

Ratiu, Völter, and Pavletic [10] find semantic tests expensive and advise to adjust their cost to the needs of the domain. In the end, writing assertions or writing properties relating input and output amounts to defining a partial formal semantics for the code generator, and this can get arbitrarily complex.

Test coverage. Test suite quality. If you used a reference example to drive the implementation of a generator, there is a risk that the design is overfitted to this single example. Mitigate this risk by testing with other models, and using coverage criteria to assess the diversity of your test suite. We recommend covering all the elements in the input meta-model and all the code generation rules (the code of the generator).

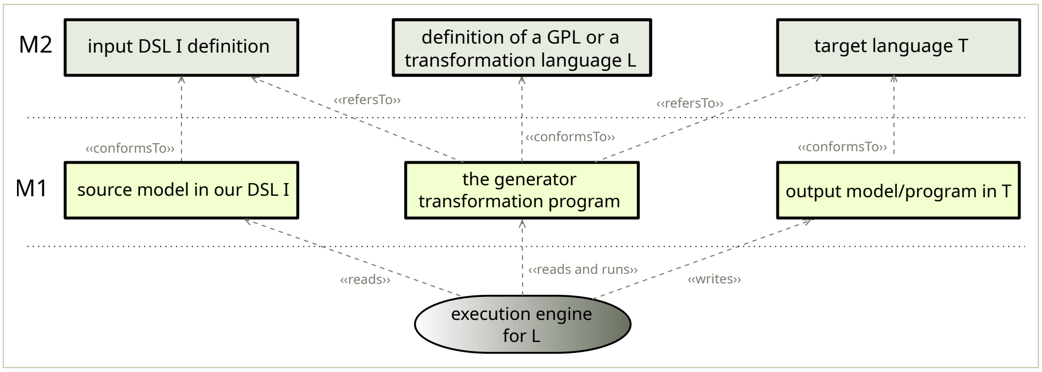


Figure 9.15: *Architecture of a code generator. Three languages are involved: input language I, transformation language L, and target language T. Each of these can, but do not have to, be different from the others. The figure is adapted from Fig. 7.10 for code generation*

For many of the above testing patterns, we need to generate random inputs. We can use input models in either abstract syntax or concrete syntax. Abstract-syntax models are relatively easily created using generator frameworks of the property-based testing libraries (Chapter 5). Concrete syntax models, on the other hand, are easier to create and inspect manually.

Another strategy is to start with a collection of manually created seed models and *mutate* them automatically by introducing small changes (shifting parts of the model, removing parts, changing operators, renaming objects, etc.). This requires implementing a mutator that executes small M2M transformations before we invoke the generator. A mutator for testing generators must preserve static correctness of the model it is mutating, as we typically do not want to generate code from invalid input models.

9.8 Code Generation in the Language-Conformance Hierarchy

Figure 9.15 gives a high-level view of a code generation transformation. We have an input in some source language I, typically an abstract syntax of our DSL, but could also be a pre-transformed input. The input conforms to the language definition for I. Furthermore, a transformation language L is used to implement a code generator. The code generator program also conforms to L's definition—otherwise we would not be able to execute it. When the generator is complex, several languages can be involved. For `robot`, we have used Python and Jinja (itself an external DSL). Finally, the generator produces an output in some target language T. This output *should* conform to some output meta-model for T, but most practically used generators do not guarantee this. Like in all our examples, most often the generator program does not refer to the output language definition, it simply produces text, and we need to use testing to establish the conformance of the output. So in practice, the two arrows in the top right corner of the figure are not guaranteed to be enforced by construction.

The choice of the two languages, the generator language (L) and the target (T), is obviously an important decision in planning the entire DSL imple-

Natural Targets for Code Generation

The following table lists target languages for code generation to inspire your own designs. This taxonomy is obviously incomplete, but we hope you will extend it with your experiences.

target group | **some examples**	**applications**
low-level programming languages | C, C++, Rust	Used for generating controllers for embedded systems, hardware drivers, operating system components. Compilers for C exist for almost any hardware platform conceivable. Low-level programming languages are also an important target for generators where performance and parallelization is of high importance (statistical software, machine learning).
business application platforms | JVM (Java, Scala, Kotlin), .NET (C#, F#)	These languages are natural targets for generating applications based on large object-oriented enterprise frameworks. Such generators can realize logic of tailored applications, business processes, GUIs, etc.
web programming | JavaScript, TypeScript, HTML, CSS, PhP	Generation of web front-ends or their customizations, generating customizations of web-sites for language, browser, connection speed, viewing platform (including mobile phone apps). Templates are very popular in web space. Since JavaScript is also executable on servers, it supports a growing number of business software back-ends, similar to the applications for generators targeting Java and .NET virtual machines.
scripting languages | Python, Shell, Ruby, Groovy, Lua	Scripting languages are a natural target for DSLs that describe packaging and deployment of software and automation tasks in sysadmin.
structured data | YAML, JSON, XML	When the goal is to convert the information from a DSL model to a format that can be easily read on another platform, a simple generator can pretty-print to a structured widely supported format like YAML. A DSL can also be used to provide a customization interface for a complex software system and the generator creates a configuration file, often in a structured file format.
logic and constraint programming and optimization | SAT (DIMACS,KodKod), Alloy, Z3, Prolog, linear programming toolkits	When your DSL defines a constraint problem, and its semantics involves finding a (possibly optimal) solution, it is natural to reduce the semantics to an existing solver or a Monte Carlo inference language. Examples: real-time scheduling, controller synthesis, UI layout, staff roster, timetabling, test case synthesis, etc.
software analysis tools | Spin, Uppaal, Simulink Design Verifier, KeY, Coq, Agda	There are many analysis tools for models of diverse complexity. If the purpose of the DSL is to create an analysis case, one often translates the DSL models into input models for an analysis tool. In this case, the target language is often a DSL, too.

mentation. Unlike for interpreters, `T` and `L` can be different for generators, and often are. The generator language `L` should be selected in agreement with the rest of the language tool chain (parsing, meta-modeling, static semantics). `L` should support the infrastructure that is needed to generate code such as templates, but also have sufficient expressiveness to do complex calculations and to create data structures. On the other hand, the target language `T` should be selected by the requirements of the *target* platform, so the execution environment for the created code. If this environment requires performance and memory efficiency, C or Rust might be good choices. If this environment is a business application framework, Java might be a good choice. We discuss some of these decisions in the info box above.

Crucially, the target code does not have to maintain any dependencies on the input model and the associated DSL infrastructure. Indeed, being able to deploy on specialized platforms, without any dependencies on our development environment is one of the key reasons to use a code generator. Nevertheless, if there are no user requirements to separate the target language from the generator language, one should probably not do so. Using the same language allows reuse of expertise in the project and makes the code generator easier to maintain. The same programmers that develop the system linked against the generated code can then potentially maintain the generator. For `prpro` we have used a different source, target, and implementation language, while for `robot` we used Python both in the generator and in the target.

Further Reading

Kahani et al. [9] and Czarnecki and Helsen [4] survey M2T transformation languages and classify them in more detail than we did here. A large number of integration patterns for the static and generated code, along with an evaluation of their requirements, are described by Greifenberg et al. [5, 6].

Pretty-printers are programs that serialize abstract syntax to concrete syntax. Pretty-printers are M2T transformations, and the simplest kinds of code generators; here the source and the target language are the same, just in different representations. The popular compositional design of pretty-printers is due to Hughes and Wadler [7, 13]. The `paiges` library chapter follows this design. The design has proven useful not just for pretty-printing, but for formatting structured ASCII text in general, including in code generation. Implementations exist for many languages.

Binding-time analysis discussed in Guideline 9.4 was originally developed for automatic derivation of generators from interpreters [8]. While automatic derivation of generators is a difficult problem, still far from language-engineering practice, binding-time analysis provides a systematic basis for manual design of generators.

One key advantage of M2M transformations is that they can guarantee type correctness of the output statically. This means that a transformation will always produce correct output models. This is much harder to ensure for M2T transformations, including generating code. The existing attempts require a grammar for the target language and can guarantee that the output is parseable [12, 3]. This introduces cost to M2T, getting it closer to M2M, but still allows some illusion of a template to be maintained. Probably the most attractive language in this space is TXL [3].

The report of Ratiu, Völter, and Pavletic [10] is a detailed source of inspiration on aspects of testing DSL implementations, including non-trivial ideas and experiences regarding code- and test-generation, richer than what Sect. 9.7 reports.

Additional Exercises

Exercise 9.8. Create a code generator translating the `fsm` language to C by modifying the Java code generator found in fsm.xtend/src/main/xtend/dsldesign/fsm/xtend/ToJavaCode.xtend. Use any template language you find interesting to study.

Exercise 9.9. Implement a graph visualizer for `fsm`. Graphviz (http://graphviz.org) is a graph visualization tool which comes with its own DSL for describing graphs. Use the DSL to draw graphs of finite-state machines. The generator should produce a Graphviz file, and then use Graphviz to lay the graph out in an image.

Exercise 9.10. Implement a mutator for `fsm` instances: an M2M transformation that randomly performs a small change to a state name, modifies a connection of a transition, drops or adds a transition, or changes which state is initial. Use this mutator to test robustness of our `fsm` generator. Try to assess the achieved test coverage.

Exercise 9.11. Implement a code generator for `petrinet` (Fig. 7.3, p. 239).

Exercise 9.12. Let our input language be the CSV format (comma-separated-values), with the first row containing distinct column names. Find a CSV handling library for a programming language of your choice, and use it to load a file. Then implement an M2T code generator into YAML, JSON, or XML (pick one), translating each row to a single object, with column names being the field names (tag or attribute names for XML). Do not use any library handling YAML/JSON/XML.

Exercise 9.13. Reimplement the above generator using an M2M transformation: design a simple meta-model or ADTs to represent flat objects, convert the data to this AST first (M2M), and then serialize it using a pretty-printer of your own design (M2T). Alternatively, use an existing library handling YAML/JSON/XML, and build an instance of its representation using an M2M transformation, then serialize using the library's pretty-printer. Reflect on the advantages and disadvantages of the two strategies from this and the previous exercise.

Exercise 9.14. Implement a generator of random CSV files (or find one), and use it to test robustness of a generator from one of the two exercises above.

Exercise 9.15. Reimplement the code generator for `prpro` expressions (Fig. 9.3) to use `java.lang.StringBuilder` instead of the naive concatenation of strings.

Exercise 9.16. Use the pretty-printing library (`paiges`) to improve the generator for `prpro` from Fig. 9.7. Ensure that the code generated is at most 80 columns wide and is properly indented according to Python rules, even if some lines wrap. The wrapping of the generated expression code is controlled in Line 26 of this figure, where it is presently set to `-1`, in order to make all line lengths unlimited.

Exercise 9.17. Reimplement the code generator for `prpro` using the Pyecore library and the Jinja template language, like we did for `robot` in this chapter.

Exercise 9.18. Our generators for `prpro` do not sanitize variable names. Speculate what could happen if variable names stored in the XMI input file contained single quotes, double quotes, and then arbitrary strings? Adapt the code generator of Figures 9.5 and 9.7 to escape single quotes, either using your own custom sanitization function or (better) using an existing sanitization library available for JVM.

Exercise 9.19. Write a transformation that translates instances of the meta-model `classmodel` presented in Fig. 9.16 to Java types. All names in the input models are unique. All attributes are integer valued for simplicity. This way we do not have to represent types in the meta-model. You can find a Scala ADT version of the same meta-model at classmodel.scala/src/main/scala/dsldesign/classmodel/scala/syntax.scala if you prefer to work on this exercise in a functional style. Notice that this generator is a smaller sibling of the official Ecore generator that we have been using all the time to obtain the implementations of our meta-models in Java.

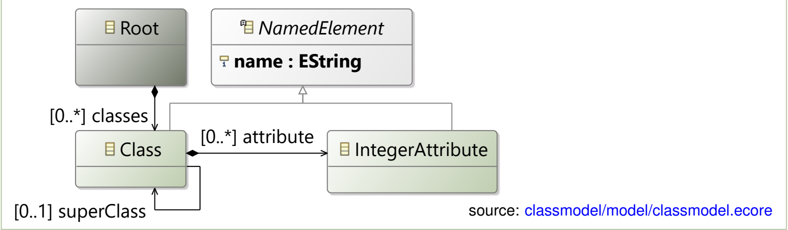


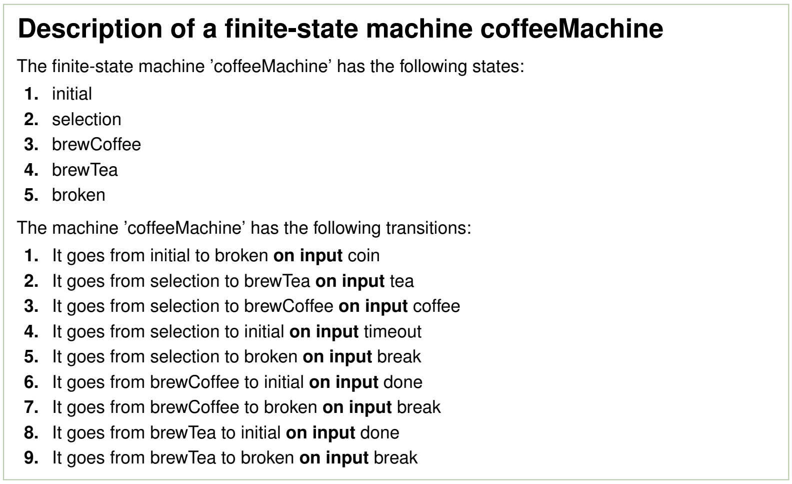
Figure 9.16: A simple class diagrams meta-model, `classmodel`, with class generalization and integer attributes, but no associations

Exercise 9.20. Implement a generator of random instances for the meta-model from the previous exercise and use it to test structural correctness of the Java files created by your code generator.

Exercise 9.21. Implement a generator for `fsm` that produces a readable summary of a machine to be displayed in a web browser (generate the suitable HTML file). A simple instance of `fsm` representing a coffee machine is shown in Fig. 9.17 (see also Fig. 7.2 and the meta-model in Fig. 3.1). An example rendering for this instance is shown in Fig. 9.18. (See also Exercise 7.10.)

```
machine CoffeeMachine [

  initial ^initial

  state ^initial [
    on input "coin"  output "which drink do you want?"     and go to selection
    on input "break" output "machine is broken"            and go to broken
  ]

  state selection [
    on input "tea"     output "serving tea"                and go to brewTea
    on input "coffee"  output "serving coffee"             and go to brewCoffee
    on input "timeout" output "coin returned; insert coin" and go to ^initial
    on input "break"   output "machine is broken!"         and go to broken
  ]

  state brewCoffee [
    on input "done"    output "coffee served. Enjoy!"      and go to ^initial
    on input "break"   output "machine is broken!"         and go to broken
  ]

  state brewTea [
    on input "done"    output "tea served. Enjoy!"         and go to ^initial
    on input "break"   output "machine is broken!"         and go to broken
  ]

  state broken
]
```

Figure 9.17: An example instance of an `fsm` in textual concrete syntax for the coffee machine model shown previously in graphical syntax in Fig. 7.2 on p. 238



Description of a finite-state machine coffeeMachine

The finite-state machine 'coffeeMachine' has the following states:

1. initial
2. selection
3. brewCoffee
4. brewTea
5. broken

The machine 'coffeeMachine' has the following transitions:

1. It goes from initial to broken **on input** coin
2. It goes from selection to brewTea **on input** tea
3. It goes from selection to brewCoffee **on input** coffee
4. It goes from selection to initial **on input** timeout
5. It goes from selection to broken **on input** break
6. It goes from brewCoffee to initial **on input** done
7. It goes from brewCoffee to broken **on input** break
8. It goes from brewTea to initial **on input** done
9. It goes from brewTea to broken **on input** break

Figure 9.18: An HTML report created from the fsm instance representing the coffee machine model in Figures 7.2 and 9.17

Exercise 9.22. Implement a code generator for `expr` models (Chapter 8) to a language that does not have infix operators, for instance Racket, Lisp, or PostScript.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985 (cit. p. 321).

[2] J. Craig Cleaveland. "Building application generators". In: *IEEE Software* 5.4 (1988), pp. 25–33 (cit. p. 344).

[3] James R. Cordy. "The TXL source transformation language". In: *Sci. Comput. Program.* 61.3 (2006), pp. 190–210 (cit. p. 351).

[4] K. Czarnecki and S. Helsen. "Feature-based survey of model transformation approaches". In: *IBM Syst. J.* 45.3 (2006), pp. 621–645 (cit. pp. 331, 351).

[5] Timo Greifenberg et al. "A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Ed. by Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe. SciTePress, 2015 (cit. p. 351).

[6] Timo Greifenberg et al. "Integration of handwritten and generated object-oriented code". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Springer. 2015 (cit. p. 351).

[7] John Hughes. "The design of a pretty-printing library". In: *1st International Spring School on Advanced Functional Programming Techniques*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. LNCS. Springer, 1995 (cit. p. 351).

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Inc., 1993 (cit. pp. 342, 351).

[9] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. "Survey and classification of model transformation tools". In: *Software & Systems Modeling* 18.4 (2019), pp. 2361–2397 (cit. p. 351).

[10] Daniel Ratiu, Markus Völter, and Domenik Pavletic. "Automated testing of DSL implementations—experiences from building mbeddr". In: *Software Quality Journal* 26.4 (2018), pp. 1483–1518 (cit. pp. 347, 348, 351).

[11] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005 (cit. pp. 341, 345).

[12] Guido Wachsmuth. “A formal way from text to code templates”. In: *Fundamental Approaches to Software Engineering (FASE)*. 2009 (cit. p. 351).

[13] Philip Wadler. “A prettier printer”. In: *The Fun of Programming*. Ed. by Jeremy Gibbons and Oege de Moor. Bloomsbury, 2003 (cit. p. 351).

Abstraction is the most important factor in writing good software. (...) A domain-specific language is the "ultimate abstraction."

Paul Hudak [6]

10 Internal Domain-Specific Languages

In the previous chapters, we have focused on the construction of *external domain-specific languages*. Their development follows a compiler-like pipeline architecture, with clearly separated design artifacts: concrete and abstract syntax, types and constraints, an interpreter or a generator. Building external DSLs might feel like reimplementing large parts of GPL functionality, especially if your DSL includes logic or expressions, or you need an editor with support for static checking, code completion, etc. This should not be necessary, given that excellent implementations of GPLs providing this functionality are readily available. In this chapter, we investigate the design and implementation of *internal domain-specific languages*, which, implemented as libraries in another language, are able to reuse the infrastructure of GPLs and embed well into an existing development setup.

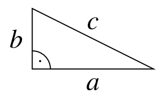
Think of an internal DSL as an idiomatic slang, a sub-language. A large language, like English, can be used to talk about anything, but an idiomatic slang focuses on efficiency for a narrow domain. For example, in a Canadian coffee shop the idiom "double double" means an order for a coffee with double cream and double sugar—quite far from the general meaning of the word "double!" Coffee shop slangs are internal DSLs embedded in the English language, but used purely to order coffee products. Similarly, in computing, internal DSLs are implemented as a specialization of a larger language to solve a specific problem efficiently, assigning new idiomatic meaning to the existing language constructs.

The key idea of internal DSLs is to exploit the syntax of a suitable GPL to create an idiomatic impression. The following polynomial equation, where α is a non-zero constant, does not seem to carry a lot of interesting meaning:

$$\alpha(x_1^2 + x_2^2) - (\alpha - 1)x_3^2 = x_3^2 \tag{10.1}$$

We transform the equation; reduce the terms with the same variable, divide by α, and rename x_1, x_2, x_3 to a, b, c respectively. This transformation, even if semantically preserving, "magically" reveals a familiar idiomatic meaning, the Pythagorean theorem:

$$a^2 + b^2 = c^2$$



Using a specific format with familiar identifiers has exposed the information much more clearly—a property of right triangles. The new meaning is an interpretation that has been added by us, the human readers, who have

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_10

```scala
val m = (state machine "coffeeMachine"
  initial "initial"
    input "coin"    output "what drink do you want?"     target "selection"
    input "idle"                                         target "initial"
    input "break"   output "machine is broken"           target "deadlock"
  state "selection"
    input "tea"     output "serving tea"                 target "making tea"
    input "coffee"  output "serving coffee"              target "making coffee"
    input "timeout" output "coin returned; insert coin"  target "initial"
    input "break"   output "machine is broken!"          target "deadlock"
  state "making coffee"
    input "done"    output "coffee served. Enjoy!"       target "initial"
    input "break"   output "machine is broken!"          target "deadlock"
  state "making tea"
    input "done"    output "tea served. Enjoy!"          target "initial"
    input "break"   output "machine is broken!"          target "deadlock"
  state "deadlock"
end)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep/coffeeMachine.scala

Figure 10.1: A coffee machine model in the internal DSL variant of the finite-state-machine language, embedded in Scala

context to interpret the syntax when they recognize familiar idioms. This is the main idea of internal DSLs: use a general-purpose programming language that can express arbitrary computations, and provide a vocabulary (an API) that allows the users of the language to interpret the program text abstractly—as a model, often resembling some other familiar notation.

Definition 10.2. *An* external *DSL is defined and designed separately from any GPL. Its standalone implementation provides its own concrete syntax (parser), static semantics, and a back-end (execution mechanism).*

Definition 10.3. *An* internal *DSL is implemented as a library within a* host *GPL. Its models are programs in the GPL. An internal DSL reuses the concrete syntax along with the basic static and execution semantics of the host.*

We have already seen examples of internal DSLs in earlier chapters. The most prominent one was the language of parser combinators used to build concrete syntax in Chapter 4 (see Fig. 4.7 and the info box on page p. 113). In these examples, parser combinators are themselves internal DSLs hosted in Scala and Haskell. The models built in this language are GPL programs, but it is more natural to read them as if they were formal grammars.

In this chapter, we shall learn the key patterns of implementing internal DSLs: the deep and shallow embedding. Then we discuss a range of advantages, use cases, and design guidelines for internal DSLs. We visit a zoo of examples of practical internal DSLs, we consider testing such DSLs, and position them in the framework of the meta-modeling hierarchy of Chapter 3.

10.1 Internal DSLs with the Deep Embedding Pattern

Figure 10.1 shows an example state machine model of a coffee machine written in an internal DSL hosted in Scala. It is useful to compare this model with Fig. 4.5 on p. 102, which shows another model in an external

```
     model → 'state' 'machine' String state* 'end'
     state → ( 'state' | 'initial' ) String transition*
transition → 'input' String ( 'output' String )? 'target' String
```

Figure 10.2: A simple context-free grammar for `fsm`*, in preparation for designing an internal DSL*

```
               model → 'state' machine
             machine → 'machine' String initialOrStateOrEnd
initialOrStateOrEnd → 'state' String inputOrNextState
                     | 'initial' String inputOrNextState
                     | 'end'
    inputOrNextState → 'input' String outputOrTarget
                     | 'state' String inputOrNextState
                     | 'initial' String inputOrNextState
                     | 'end'
      outputOrTarget → 'output' String target
              target → 'target' String inputOrNextState
```

Figure 10.3: The grammar from Fig. 10.2 transformed to a simple prefix form that is easy to implement as an object-oriented API

DSL variant of `fsm`. First, perhaps the most striking difference between the two designs of syntax is that in Fig. 10.1, a model is a fragment of a Scala program, an expression calculating an AST, subsequently bound to immutable variable `m`. This discloses that the model is really just an idiomatic Scala program. Second, we no longer use square brackets to denote blocks. This is because Scala reserves them to mark generic type parameters and we cannot nest code inside other than types. We often adapt to the restrictions of the host language when designing syntax for internal DSLs.

In the deep embedding pattern, we clearly separate the front-end from the back-end of the language, similarly to external DSLs. We use the host language to implement the front-end instead of an external parsing tool. In this particular example, the back-end constructs an Ecore representation, which can later be executed using interpreters and code generators as shown in Chapters 8 and 9. The construction of the abstract-syntax is the essence of the deep embedding pattern, but not the use of Ecore. Other representations can be used, most typically the plain types of the host language.

The front-end of a deep internal DSL parses the input and constructs the abstract syntax simultaneously. For pedagogical reasons, we explain the two activities separately, starting with parsing. We begin by sketching a simple context-free grammar for the `fsm` language, with one machine per model for simplicity.[1] See Fig. 10.2.

We will use a few steps to implement an API (in Scala) which uses types to constrain legal sequences of calls, effectively allowing the writing

[1]Building internal DSLs from grammars is not recommended. It is often impossible to realize a grammar precisely in the host GPL. We do it here to make the presentation more precise.

of sentences derived from the grammar in Fig. 10.2. We first turn the grammar into a form where each production expands to a keyword terminal, possibly followed by a parameter value and a non-terminal responsible for the continuation, the “what-comes-after.” The entire transformed grammar is presented in Fig. 10.3. Convince yourself that it generates the same language as the one we started with. The rewriting has “chopped off” every production after one or two terminals and introduced a new non-terminal for the tail. This form is quite easy to map to an object-oriented API. A non-terminal becomes a class, and each keyword is implemented as a method, with the subsequent value becoming the argument for a call.

Figure 10.4 shows an encoding of this grammar as a Scala API. We open with the named value (object) `state`; just referring to `state` creates an illusion of using a keyword in the internal DSL, even though this is just a value exposed by our API. The methods in the `state` object implement the keywords in the next production, here the machine non-terminal. The method takes the name of the machine as an argument, so a string value can follow in our “grammar,” and returns the object representing initialOrState-OrEnd. The return type provides three methods as there are three possible expansions for this production: starting with `'state'`, `'initial'`, and `'end'`. The first introduces a state, the second introduces an initial state, and the third one just closes the machine. The implementation returns a unit value, as we are purely concerned with parsing for now. This scheme to obtain the implementation from a grammar is not general, yet it works for any grammar of this shape, and for any object-oriented host language.

Exercise 10.1. Write a small model of a state machine using the API of Fig. 10.4. For instance a machine with a single state and a loop transition. Understand the types of all sub-expressions in the program.

Figure 10.5 recreates fragments of the coffee machine model, annotating types for parts of the model. The example `t0` simply refers to the `state` object predefined in Fig. 10.4; it returns the singleton object itself. In the `t1` expression we call its only method, `machine`. Here we exploit a bit of syntactic flexibility in Scala: `state.machine ("coffeeMachine")` can be written without the navigation operator (dot) and without parentheses around the argument; any unary method can be invoked as if it was an infix operator. Compare also the fragment `t4` with `t4withDots` for a larger example of the same. Appreciate how layout changes (unconventional line breaks and indentation) create an illusion of a DSL out of a Scala expression.

A dot-free parentheses-free syntax, like above, should be possible to attain in any GPL that provides the ability to define your own infix operations. In a host language where dots and parentheses are unavoidable, we would use the following design, known as a *fluent interface*:

```
state .machine ("coffeeMachine")
  .state ("initial")
.end ()
```

```
 1 object state:
 2   def machine (name: String) = INITIAL_OR_STATE_OR_END // Allow opening keywords: "state machine"

 4 object INITIAL_OR_STATE_OR_END:                     // In a machine context, open a state block
 5                                                     // using "state"/"initial", or close with "end"
 6   def state (src: String) = INPUT_OR_NEXT_STATE     // Open a non-initial state. An "input", a new
 7                                                     // state, or "end" may follow.
 8   def initial (src: String) = INPUT_OR_NEXT_STATE // Open a new initial state.
 9   val end = ()                                      // Close the current machine object.

11 object INPUT_OR_NEXT_STATE:                         // A new transition or close a state
12   def input (input: String) = OUTPUT_OR_TARGET      // Parse a new transition definition
13   def state (name: String) = INITIAL_OR_STATE_OR_END.state (name)      // End a state definition,
14                                                                        // start a new state
15   def initial (name: String) = INITIAL_OR_STATE_OR_END.initial (name) // End a state definition,
16                                                                        // start an initial state
17   val end = ()                                      // Close a machine. Return to top level

19 class TARGET:                                       // Detect the target state phrase
20   def target (name: String) = INPUT_OR_NEXT_STATE // Detect "and go" and then await for "to"

22 object OUTPUT_OR_TARGET extends TARGET:   // Detect an (optional) output or a target (inherited)
23   def output (output: String) = TARGET () // Record a transition output  and move to new state
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep/justParsing.scala

Figure 10.4: *The implementation of parsing for the finite-state-machine internal DSL in Scala. See full version in Fig. 10.7*

The complete implementation needs to construct an abstract syntax of the state-machine model on the fly, while the internal DSL expression is evaluated. We will do this by extending Fig. 10.4 with suitable computations. While "parsing" we receive the information from the user in small pieces, so we need a representation that can be updated incrementally. Figure 10.6 shows a simple incrementally updatable representation, which aggregates the name of the state machine, state names, transitions, and the name of the initial state (when available, thus an `Option`). For each transition we store the name of the source state, and add the input and output labels, with the name of the target state, when they become available.

Figure 10.7 shows the implementation of the internal DSL for finite-state machines, including the construction of an abstract-syntax value. Whereas in Fig. 10.4 the second line delegates to a simple object implementing parsing past the opening keyword, in Fig. 10.7 we delegate to a class encapsulating a model representation object. The rule in line 2 invokes the constructor `ModelRep` to store the machine name in the parser state. This object is then available for the subsequent rules. For instance, in line 6 we update the parsing state with the name of the initial state and in line 8 add a new state name to a list. In contrast, in Fig. 10.4 the corresponding lines (still 6 and 8) do not pass any values but just delegate to the tail parsing object.

Line 9 in both figures implements the finalization of the machine. In the extended version we pass the constructed model object to a helper function (`modelRep2Model`) that implements a model-to-model transformation from Scala object spaces to an Ecore representation. In the corresponding line in

```scala
val t0: state.type =
  (state)

val t1: INITIAL_OR_STATE_OR_END =
  (state machine "coffeeMachine")

val t2: unit =
  (state machine "coffeMachine"
   end)

val t3: INPUT_OR_NEXT_STATE =
  (state machine "coffeeMachine"
     initial "initial")

val t4: unit =
  (state machine "coffeeMachine"
     initial "initial"
  end)

val t4withDots: unit = // same as above but with explicit dot and parens
  state
    .machine ("coffeeMachine")
    .initial ("initial")
    .end

val t5: OUTPUT_OR_TARGET =
  (state machine "coffeeMachine"
    initial "initial"
      input "coin")

val t6: TARGET =
  (state machine "coffeeMachine"
     initial "initial"
   input "coin" output "what drink do you want?")

val t7: INPUT_OR_NEXT_STATE =
  (state machine "coffeeMachine"
    initial "initial"
      input "coin" output "what drink do you want?" target "selection")

val t8: unit =
  (state machine "coffeeMachine"
    initial "initial"
      input "coin"  output "what drink do you want?" target "selection"
  end)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep/coffeeMachineExplained.scala

Figure 10.5: Fragments of the coffee machine model with type annotations, demonstrating how the syntax is parsed (typed). Especially note `t4withDots` *to appreciate how the ability to drop parentheses from method arguments and dot from method calls in Scala helps to create an illusion of another language*

Fig. 10.4, we have simply returned a unit value, so basically ‘nothing.’ The use of Ecore here is inessential: it is just a model-to-model transformation injected in the final step. It could have been replaced with whatever else you need: performing cleanup, expanding syntactic sugar, static checking, converting to other formats, etc.

Observe how concise the language implementation is. Figures 10.6 and 10.7 contain almost all the code; everything but the final transformation is included. This code arranges parsing and basic type checking. Still,

```scala
sealed case class ModelRep (
  name: String,
  states: List[String] = Nil,
  tran: List[TranRep] = Nil,
  initial: Option[String] = None )
sealed case class TranRep (
  source: String,
  input: Option[String] = None,
  output: Option[String] = None,
  target: Option[String] = None )
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep.scala

Figure 10.6: *Data types for collecting information during parsing in a deeply embedded implementation of* `fsm` *(a temporary abstract syntax)*

```scala
1  case object state:
2    def machine (name: String) = INITIAL_OR_STATE_OR_END (ModelRep (name))

4  case class INITIAL_OR_STATE_OR_END (machine: ModelRep):
5    def initial (src: String) =
6      INPUT_OR_NEXT_STATE (machine.copy (initial = Some (src), states = src ::machine.states), src)
7    def state (src: String) =
8      INPUT_OR_NEXT_STATE (machine.copy (states = src ::machine.states), src)
9    def end: fsm.Model = modelRep2Model (machine)

11 case class INPUT_OR_NEXT_STATE (machine: ModelRep, src: String):
12   def input (input: String) =
13     OUTPUT_OR_TARGET (machine, src, TranRep (source = src, input = Some (input))
14   def state (name: String) = INITIAL_OR_STATE_OR_END (machine).state (name)
15   def initial (name: String) = INITIAL_OR_STATE_OR_END (machine).initial (name)
16   def end: fsm.Model = INITIAL_OR_STATE_OR_END (machine).end

18 class TARGET (machine: ModelRep, src: String, tran: TranRep):
19   def target (name: String) =
20     val tran1 = tran.copy (target = Some (name))
21     INPUT_OR_NEXT_STATE (machine.copy (tran = tran1 ::machine.tran), src)

23 case class OUTPUT_OR_TARGET (machine: ModelRep, src: String, tran: TranRep)
24   extends TARGET (machine, src, tran):
25   def output (output: String) =
26     val tran1 = tran.copy (output = Some(output))
27     TARGET (machine, src, tran1)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep.scala

Figure 10.7: *An implementation of the* `fsm` *internal DSL: the code from Fig. 10.4 extended with construction of the abstract-syntax tree. All information is collected in a simple abstract syntax, shown in Fig. 10.6, and then, in Line 9, converted to Ecore using a helper function (not included, but see fsm.scala/ src/ main/ scala/ dsldesign/ fsm/ scala/ internal/ deep.scala)*

the obtained DSL is very flexible. For instance, we can replace any state name with a Scala expression generating the name, or we can use string interpolations. The host language provides extensibility and expressiveness.

Let us define the deep embedding pattern, and summarize the main techniques used above to implement the `fsm` language.

Definition 10.4. *A* deep embedding *is a language implementation pattern in which the elements of the implemented language are represented as values in the implementation language (the host), and not as corresponding first-class elements of the host language. The language implementation* *constructs a value representing the abstract-syntax tree of the DSL model. The tree is subsequently transformed for static checking and possible optimization, and traversed for evaluation, like with external DSLs.* [5]

What does "not first-class" mean? It simply means that all concepts in the input language map to runtime values (second-class). If our DSL has classes, they do not map to classes in the host language, but to objects (values). Similarly, constants become not the constant literals in the host, but values (objects) representing constants, etc.

The main ideas used here to realize parsing were:

- *Static values* (singleton objects) and *method names* for introducing keywords in the language,
- *Classes* for controlling which names are allowed in a context,
- *Method arguments* for introducing free values (could be any expressions) like identifiers and references,
- A *context object in the parser storing the AST*, adjusted and passed forward from construct to construct. For this simple example, we used a simple flat value. If you need nested structures (recursion in the grammar), the context object should embed a stack or a tree, very much like with generated parsers and PEGs.

Exercise 10.2. Modify the implementation of the deeply embedded internal DSL, so that instead of the keyword `end`, we use parentheses to enclose states nested in a machine, for example:

```
val m = state machine "coffee Machine" (
  initial "initial"
    input "coin" output "what drink do you want?" target "selection"
    ...
)
```

We can also modify the state syntax consistently so that it uses parentheses as well. This is a non-trivial change, as `initial` can no longer be a method on the object constructed by `machine`. Instead `machine` can be a curried method that takes two arguments, and the parentheses are placed around the second one. (Unfortunately, the same seems impossible with braces in Scala.)

Context Objects

When building the abstract-syntax tree above, we maintain the parser state in a context object representing a partial AST (`ModelRep`). In a pure program, we have to construct a new augmented context object whenever new information becomes available (e.g., `TranRep(source=src,input=Some(input)`, line 13 in Fig. 10.7). In imperative (impure) programs, there is no need to create new objects. The existing context object can be updated directly. We demonstrate this with a similar DSL implemented in Python.

As usual we begin with defining the abstract syntax. Figure 10.8 defines a simple representation for transitions and models. A `Model` (Line 16) has a name, an initial state name, and a list of transitions. A transition (`Tran`,

```
 1 class ModelElement:
 2     """Shared functionality between meta-model elements"""
 3     def __repr__(self):
 4         """Support serialization for print-debugging and tests"""
 5         return str(self.__class__) + ": " + str(self.__dict__) + "\n"

 7 class Tran(ModelElement):
 8     def __init__(self: Tran, source: str, input: str, output: str, target: str) -> None:
 9         # These assertions will catch some syntax errors (at runtime)
10         assert source != "" and input != "" and target != ""
11         self.source = source
12         self.input = input
13         self.target = target
14         self.output = output

16 class Model(ModelElement):
17     """A simple Python abstract syntax (a meta-model) for FSM"""
18     def __init__(self: Model, name: str) -> None:
19         self.name = name
20         self.initial: str = ""
21         self.tran: List[Tran] = []

23     @property
24     def states(self: Model) -> Set[str]:
25         """A derived property listing all states. Should only be used after the
26            Model is constructed."""
27         return {t.source for t in self.tran} | {t.target for t in self.tran}
```
source: fsm.py/FsmInternalDeep.py

Figure 10.8: *A simple meta-model (abstract syntax) for finite-state machines in Python*

Line 7) has a source state, an input label, a target state, and an output label. States are represented just by their names as character strings.

> **Exercise 10.3.** Write down the coffee machine of Fig. 10.1 using this abstract-syntax (meta-model) in Python.

Figure 10.9 presents our coffee machine model in an internal DSL of Python (the implementation of the DSL will be shown below). In order to show different means available to Python programmers, we include four variants of this DSL, all based on the same abstract syntax. Since all these variants are compatible, we mix them in a single model, a different style for each state. We begin with Python's idiomatic dictionaries. The `step` function creates a transition based on the received dictionary describing a transition (lines 2–6). The second variant uses named parameters instead. The function is called `transition` to avoid a name clash with `step` (lines 8–12). The third variant uses a fluent interface style, very much like in Scala, but with an explicit navigation symbol (lines 14–17, see also p. 360). Unlike in Scala, we have to use explicit navigation (the 'dot' operator), because Python methods are not infix operators. One could override infix operators in Python (see https://docs.python.org/3/library/operator.html#mapping-operators-to-functions). Variant IV (lines 19–21) exploits this to separate transition elements with operators. The last variant creates the strongest illusion; how-

```
 1 with Model("coffeeMachine") as m:
 2   # Variant I: Dictionary-based
 3   step({"source": "initial", "input": "coin",
 4         "output": "what drink do you want?", "target": "selection" })
 5   step({"source": "initial", "input": "idle", "target": "initial" })
 6   step({"source": "initial", "input": "break", "output": "machine broken", "target": "deadlock" })

 8   # Variant II: Keyword arguments-based
 9   transition(source="selection", input="tea", output="serving tea", target="makingTea")
10   transition(source="selection", input="coffee", output="serving coffee", target="makingCoffee")
11   transition(source="selection", input="timeout", output="coin returned", target="initial")
12   transition(source="selection", input="break", output="machine is broken!", target="deadlock")

14   # Variant III: Fluent style
15   state("makingCoffee") \
16       .input ("done").output ("coffee served. Enjoy!").target ("initial") \
17       .input ("break").output ("machine is broken!").target ("deadlock")

19   # Variant IV: Operator style
20   state("makingTea") ** "done"  % "tea served. Enjoy!" >> "initial"
21   state("makingTea") ** "break" % "machine is broken!" >> "deadlock"

23   # Designate the initial state (we use it with all the styles above)
24   initial ("initial")
```

source: fsm.py/CoffeeMachineDeep.py

Figure 10.9: *An example model of a coffee machine, demonstrating several styles of DSL design patterns in Python*

ever the first three designs are more often found in the Python ecosystem. Overloading operators is more typical of functional programmers.

All these variants have one thing in common: they access a context object throughout (`m`, line 1). The use of a context object and the `with` statement is a common pattern in Python libraries. After the scope is exited, the resulting model will be bound to the variable `m`. From then on, we can continue working with it, just like with an instance of an external DSL.

Figure 10.10 shows the implementation of all four variants. We begin with enriching the `Model` class with a context management API, so that we can use it as an argument in a `with` statement. This requires implementing functions `__enter__` and `__exit__`, respectively to set up the model and to finalize the scope exit. We add a static property `__contexts`, a stack with the current model placed at the top. Our implementation of the concrete syntax will add elements to the model on top of this stack. Using a stack allows the `with` statement to be nested. If this is not desirable, a simple value should be used, instead of a stack. The four variants are implemented in lines 16–48. The second variant, in l. 22–24, is probably the easiest to understand. When a transition is created, the values for source, input, output, and target are extracted from the keyword arguments (output is optional). Then we construct the transition object (Line 23) and add it to the model on top of the context stack. The variant I implementation (l. 19–20) simply delegates to Variant II, by reinterpreting the received dictionary as keyword arguments.

The fluent variant (line 26–44) is much more complex. Following the same pattern as in Scala, the `state` factory creates a builder object, which

```python
class Model(ModelElement):
    """(See Fig. 10.10 for the first part of the class definition)
       Support meta-models as context objects (with):"""
    __contexts: List[Model] = []
    @classmethod
    def context (cls) -> Model:
       return cls.__contexts[-1]
    def __enter__(self: Model) -> Model:
        self.__class__.__contexts.append (self)
        return self
    def __exit__(self, exc_type, exc_value, traceback) -> bool:
        if exc_type != None: return False
        self.__class__.__contexts.pop()
        return True

def initial(state_name: str) -> None:                        # Mark initial state (all variants)
    Model.context().initial = state_name

def step(tran: dict) -> None:                                # Variant I: dictionary based
    transition(**tran)

def transition(**props) -> None:                             # Variant II: with keyword arguments
    t = Tran(props["source"], props["input"], props.get("output", ""), props["target"])
    Model.context().tran.append(t)

def state(name: str) -> TranBuilder:                         # Variant III: Fluent
    return TranBuilder(name)

class TranBuilder(ModelElement):
    def __init__(self: TranBuilder, source: str) -> None:
        self.__source = source
        self.reset()
    def reset(self: TranBuilder) -> TranBuilder:
        self.__input = self.__output = self.__target = ""
        return self
    def input(self: TranBuilder, input: str) -> TranBuilder:
        self.__input = input
        return self
    def output(self: TranBuilder, output: str) -> TranBuilder:
        self.__output = output
        return self
    def target(self: TranBuilder, target: str) -> TranBuilder:
        transition(source=self.__source, input=self.__input, output=self.__output, target=target)
        return self.reset()
                                                             # Variant IV: with infix operators
    def __pow__(self: TranBuilder, input: str) -> TranBuilder: return self.input(input)
    def __mod__(self: TranBuilder, msg: str) -> TranBuilder: return self.output(msg)
    def __rshift__(self: TranBuilder, target: str) -> TranBuilder: return self.target(target)
```
source: fsm.py/FsmInternalDeep.py

Figure 10.10: The implementation of three styles of internal DSLs with Python, using context objects

supports the API defining the properties of the transition. No particular order is enforced, except that `target` should be called last, as it finalizes the transition object (it delegates to variant II again).

Finally, the variant with infix operators extends the fluid interface by replacing method names with overloaded operators on `TranBuilder` objects (l. 46–48). Here it is important to choose operators of decreasing precedence or left associativity, so that a transition is evaluated from left to right:

```
((state("makingTea") ** "done") % "tea served.Enjoy!") >> "initial"
```

Otherwise, if the parsing was

```
state("makingTea") >> ("done" % ("tea served.Enjoy!" ** "initial"))
```

we would need to override operators for standard classes, like strings, which is dangerous and not recommended. Adding the operators only to our `TranBuilder` guarantees that there will be no clashes with other libraries.

Exercise 10.4. Investigate how to write a transition without an output label in each of the four syntactic variants of the DSL in Fig. 10.10. Understand which implementation aspect makes the output label optional in each of the four designs.

Context objects always appear in internal DSL implementation in one form or another. They can be implicit, like in our Scala design, they can be explicitly introduced as in the Python `with` statement. Often they are passed as an argument to anonymous functions, especially if the internal DSL is implemented in functional style. An implicit value `it` (Kotlin, Groovy, and Xtend) as well as underscore or givens (Scala) can also be used to pass a value around in a non-invasive manner for the syntax.

10.2 The Shallow Embedding Pattern

The main task of the implementation of a deeply embedded DSL is to build an abstract-syntax tree of the input model. Once this is constructed, we hand it over to the later phases of the tool chain. Thus, an implementation of a deeply embedded internal DSL is a syntax-first approach. A *shallow embedding* is dual: the semantics is implemented first, directly in the language, while the abstract syntax is not represented or manipulated at all.

In this section, we implement a shallow DSL for finite-state machines as an example. As usual, we want to show you an example model in the resulting language first; see Fig. 10.11. Superficially, this model resembles the deeply embedded one a lot (cf. Fig. 10.1). The key difference, though, is that we do not need to interpret or compile the resulting model—the model is directly executable; the coffee machine of Fig. 10.11 is a self-contained executable Scala program.

Typically, we begin the design by asking what is the semantics of the model. How can it be represented in the host language? For the finite-state machines a transition function is a natural semantic model. A state is a function that given an input produces an optional output and a new state. If inputs and outputs are just character strings, then we obtain the following type:

$$\texttt{State} := \texttt{String} \to \texttt{Option[String]} \times \texttt{State} \qquad (10.5)$$

This can be implemented in a Scala trait or a class (we show a trait):

```scala
lazy val initial: State = (state
  input "coin"    output "what drink do you want?"  target selection
  input "idle"                                      target initial
  input "break"   output  "machine is broken"       target deadlock
)

lazy val selection: State = (state
  input "tea"     output "serving tea"              target makingTea
  input "coffee"  output "serving coffee"           target makingCoffee
  input "timeout" output "coin returned;insert coin" target initial
  input "break"   output "machine is broken!"       target deadlock
)

lazy val makingCoffee: State = (state
  input "done"    output "coffee served. Enjoy!"    target initial
  input "break"   output "machine is broken!"       target deadlock
)

lazy val makingTea: State = (state
  input "done"    output "tea served. Enjoy!"       target initial
  input "break"   output "machine is broken!"       target deadlock
)

lazy val deadlock: State = (state)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/shallow/coffeeMachine.scala

Figure 10.11: *The coffee machine model specified in the shallow version of the internal DSL. The model shall be executed from the* `initial` *state. There is no separate interpreter — each state is an interpreter itself. The language implementation is shown in Fig. 10.12*

```scala
1  type Step = String => (Option[String], State)

3  val state: State = new State:
4    def step: Step =
5      { (input: String) => (Some ("Unknown input msg!"), state) }

7  trait State:
8    def step: Step
9    def input (event: String): Suspended =
10     Suspended (source=this, event=event, output=None)

12 case class Suspended (source:State, event:String, output:Option[String]):
13   def output (o: String): Suspended =
14     Suspended (source, event, Some (o))

16   def target (t: => State): State = new State:
17     def step: Step = input =>
18       if input == event
19       then (output,t)
20       else source.step (input)
```

source: fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/shallow.scala

Figure 10.12: *The implementation of a shallow variant of the internal DSL for finite-state machines in Scala. Note that the figure shows the entire implementation of the parser, static checker, and interpreter for the language (minus minor boilerplate to conserve space). An example model is shown in Fig. 10.11*

```scala
trait State { def step: String => (Option[String], State) }
```

You can find the same type in Fig. 10.12 (l. 7-10); we just extracted the function type to a named type `Step`, as it is referred to several times in the implementation.

Now that we chose a type to represent a state, we can start writing examples. For instance, a state that ignores all inputs and produces no outputs can be written as follows:

```
val deadlock = new State {
   def step = (input: String) => (None, deadlock)
}
```

Once we have a proposal for a semantic representation, we want to design an API of syntactic operators, like `deadlock` above, that will enable us to build state machines, creating states and combining them into state machines. These operators become the concrete syntax for our DSL. If we wanted to follow a style to other `fsm` DSLs in the book the operators should be `input`, `output`, and `target`. Unfortunately, `input`, `output`, and `target` do not have uniform meaning. None of them carries enough information to specify an entire transition, or to create a function of type `Step`. We still need to know the target state and output to produce a transition function.

In principle, one could create a new semantic type for every new kind of information that appears. If we have an expression that represents a part of the transition it will have the same type as the transition, but all the missing information needs to be added as arguments. Then `input "coin"` could be a function which given a source, an output, and a target state produces a new output (option) and a new target state:

$$\begin{aligned}\mathtt{Input} := \mathtt{State} \times \mathtt{Option[String]} \times \mathtt{State}\\ \to \mathtt{String} \to \mathtt{Option[String]} \times \mathtt{State}\end{aligned} \qquad (10.6)$$

The arguments represent in order: (model elements) the source state, the output label, the target state, followed by (runtime elements) the runtime input label, a received runtime output, and the runtime target state resulting from executing the transition. We could build such partial types for `output` and `target` and compose them into `Steps`, but the various types quickly multiply and become overwhelming. To simplify, we can use deep embedding locally: let's store partial information as a value syntactically and generate the semantic function once we have everything we need to construct a transition.

The idea is shown in the implementation of Fig. 10.12. The `State` trait is equipped with a new method `input`, providing the keyword to initiate a transition sub-expression. The method stores the source state and the event in a value of class `Suspended`, which also provides the remaining keywords. The `output` method (line 13) collects the optional output label. The key part of the implementation is found in the `target` method (lines 16–20). When `target` is called, we have all the information to construct the transition function. We create a new `Step` object with its own `step` function. There we check whether the runtime `input` is the same as the guard on the transition. If it is, we return the output and the target state. If it is not we delegate to the source state's `step` function, to check whether another transition can match the input.

To complete the implementation, we still need a base handler representing an empty state. We place it in a static value named `state` (line 3), which conveniently provides the opening keyword for a state definition. This way, if no transition matches a runtime input, the control will eventually arrive at the default handler, which will produce the error message and loop; the state machine will deadlock, if you send it an invalid input.

With the implementation in Fig. 10.12, one can execute the model from Fig. 10.11 by calling the step function of the initial state with an input. The incantation `initial.step ("coin")` will produce:

```
(Some ("what drink do you want?"), selection).
```

Note that no external interpreter is invoked. If you want to continue interacting with the machine, you can take the returned target state (`selection`) and invoke its `step` method, and so on.

Some sub-expressions in this internal DSL produce deeply embedded syntax values (`Suspended` in Fig. 10.12), but every sentence still produces a semantic value, directly represented as a function (an object with a method).

Exercise 10.5. Consider several sub-expressions from the coffee machine model, and explain what is the type of these sub-expressions as implemented using this shallowly embedded internal DSL. In other words, produce a variant of Fig. 10.5 for the model of Fig. 10.11 with the implementation of Fig. 10.12.

Exercise 10.6. Modify the implementation of the shallow DSL in Fig. 10.12 so that instead of failing to a deadlock on an unknown message, it ignores the message and returns to the same state again (loops on any unknown message).

One way to do this is to pass the original source state to each step method, and make the base step method to loop not to itself, but to the original source state. This is a non-trivial redesign, characteristic of shallow DSLs, where a change in semantics can have far-reaching effects.

We summarize the key aspects of the shallow embeddings with a definition.

Definition 10.7. A shallow embedding *is a language implementation pattern in which the elements of the implemented language are mapped to the elements of the host language (first-class elements) that can be directly evaluated in the host language infrastructure, capturing the intended meaning of the DSL model. The abstract syntax of the model is not explicitly constructed or traversed. (cf. Gibbons and Wu [5])*

When designing a shallow embedding for an internal DSL we first want to understand what is the type of the semantic value that represents the meaning of the model. Very often, for models that represent executions and transformations, this value will be a function. Then we identify elements in the host language that corresponds to our language. For functional semantics we will often see function definitions and lambdas, but for other models it might be expressions or classes (as we see for the active record pattern in Ruby in Fig. 1.5, p. 10). For finite-state machines we exploited functions to capture the meaning of transitions and lazy references to link values into circular structures (we could have used nullary functions instead). Note that in the deeply embedded DSL, states were elements in the syntax tree. In the shallowly embedded DSL, states can be executed directly—they are small programs that produce other small programs (other states).

Typically we want to define a set of operators that allow us to construct the semantics of the model piecewise. The operators serve as syntax for the internal DSL, but their implementation captures the semantics. This means that the semantics of the language needs to be *compositional* with respect to syntax: each piece of syntax must be meaningfully interpretable, and there must be a relatively close correspondence between the DSL concepts and the constructs in the host language. In a deeply embedded DSL, we can transform the model first, if the execution semantics is far from the syntax. This cannot be done in a shallow internal DSL, because we process each syntactic element immediately.

On the other hand, we can obtain a very concise implementation of the language. The implementation in Fig. 10.12 is shorter than the deeply embedded variant of the same language. This is so even though the shallow DSL includes the entire execution semantics, while the deep one relies on an external interpreter. Deep embeddings tend to be easier to implement, especially for less experienced programmers. They follow a more systematic design, while shallow embeddings tend to slide towards wizardry. Shallow embeddings result in implementations that are leaner but harder to understand.

Shallow DSLs are elegantly structured from small bricks of behavior, while deeply embedded DSLs rely on big-steps and architectural layers (parsing, transformations, interpretation), like external DSLs. As a consequence, deep DSLs allow multiple back-ends, while replacing a back-end in a shallow DSL is practically impossible. It requires a new language implementation. On the other hand, a shallow DSLs tend to be easy to extend with a new keyword, as each keyword has a modular implementation in a single place, while in a deep DSL the implementation is scattered across phases.

10.3 Examples of Internal DSLs

We now show examples of internal DSLs hosted in many programming languages, to demonstrate the range of applications, and to inspire your own designs. For each language we show an example model, and discuss some design and implementation principles.

Parser Combinators as an Internal DSL

Figure 10.13 recalls the PEG rule matching a machine definition, using the dialect of the parboiled2 library. The rule is a Scala expression constructing a parser using operators ~, *, and ~>. The details of the implementation of each operator may be complex, but we read this expression at a much higher level of abstraction: as an EBNF production. We read the expression not as Scala, but as an internal DSL.

```
"machine" ~ EString ~ BEGIN ~
  stateBlock.* ~
  initialDeclaration ~
  stateBlock.* ~
END ~> FiniteStateMachine
```
source: fsm.scala/src/main/scala/dsldesign/fsm/scala/FsmParser.scala

Figure 10.13: A rule parsing the machine definition specified in the parboiled2 internal DSL. The entire grammar is shown in Fig. 4.7, p. 109

Most parser combinator libraries are shallowly embedded DSLs. Parboiled2, however, combines both patterns, and even includes aspects of code generation: the model evaluation happens partly at compile time thanks to the use of Scala compiler macros, a particularly complex and rare construction that enables high performance for parsing large files.

A basic semantic type in parser combinator DSLs is a parser or a rule: a function that takes an input stream and produces a value representing abstract syntax. The EBNF operators are functions that compose the parser values. Libraries in statically typed languages rely on types to ensure that the parsing rules are well-formed. The type system of the host language is put to work to implement the static semantics of the internal DSL. We have done the same in the finite-state-machine languages, ensuring that keywords are only available in certain contexts, for instance `target` cannot be used before `input`. However, parser combinator libraries go further, enforcing the types of produced objects.

To appreciate the idea, consider the combinator `optional(a)`, synonymous with the question mark in EBNF. The combinator is generic. It takes a rule `a` that produces a value of type `T` and creates a new rule producing a value of type `Option[T]`. We could state its type as follows (simplified):

```
def optional[T] (a: Rule1[T]): Rule1[Option[T]]
```

In addition, parboiled2 uses types to enforce disciplined use of the parsing stack, making sure that when a stack is reduced, it always has enough values. `Rule1[T]` is a type of a rule that pushes a value of type `T` on the stack. Since `optional(a)` pushes the same value wrapped in an option or `None`, the operator does not change the arity of the rule. Both input and output are `Rule1`. Similarly, the operator `zeroOrMore` (Kleene star) changes a `Rule1[T]` into a `Rule1[Seq[T]]`.

Finally, we remark that many internal DSLs for parsing reuse the facility of regular expression matching of the host language for matching tokens (terminal symbols). Remember that not just functions and types of the host language can be reused in an internal DSL. Anything, including important libraries, of the host language can be exploited.

Sinatra: An Internal DSL for Web Programming in Ruby

Sinatra[2] is a lightweight web framework in Ruby for building simple web apps. It is reportedly used by many companies, including GitHub, Apple,

[2]https://www.sinatrarb.com/, accessed 2022/09

Figure 10.14: *Left: A web server rendering 'Hello World' using Sinatra. Right: The corresponding Ruby code implementing the same without an internal DSL*

```ruby
require 'sinatra'

get '/hello' do
  'Hello world!'
end
```

```ruby
app = NoDSL::Application.new
app.on_request(:get, :path_info => '/hello')
  do |response|
    response.body = "Hello world."
  end
```

Figure 10.15: *A parameterized route handler in Sinatra*

```ruby
get '/hello/:name' do
  # When matching "GET /hello/foo" and "GET /hello/bar"
  # Then params[:name] is 'foo' or 'bar'
  "Hello #{params[:name]}!"
end
```

BBC, and Accenture. Sinatra's design is not unique to Ruby. The language has inspired web frameworks for other languages, such as Express for Java Script[3] and Scalatra for Scala.[4] Its API forms an internal DSL shallowly embedded in Ruby. Sinatra's syntax is build around the basic verbs in the HTTP protocol: `GET`, `POST`, and `PUT`. The left part of Fig. 10.14 shows a small piece of Ruby code using the Sinatra library. This code implements a web server that outputs a static page containing `'Hello world!'` whenever a user accesses the path `'/hello'`. The right-hand side of the figure shows the similar functionality implemented using a classic library. It is easy to appreciate that the internal DSL in the left part succeeds in hiding a lot of complexity.

In Sinatra terminology, a single block defining a reaction to a request is called a "route." The specifications of `post` and `put` routes are similar to `get`. Routes are matched in the order they are defined. The first route that matches the request is invoked. Route patterns can be parameterized. Figure 10.15 shows a parameterized version of the `get` route. If accessed with `/hello/Garfield`, the app will respond with *Hello Garfield*.

How is this implemented? The `do`–`end` block is Ruby's form of an anonymous function, which is passed as the last parameter to a function (to the `get` function in the above example). The code in the block is handled lazily. It is not executed at the time of calling `get`, but the `yield` statement can pass control to it at a suitable point, inside the `get` implementation. The `get` route is basically a function (method) that matches its argument pattern to an incoming request, binds parameters to values during this matching, and yields to the block provided within `do`–`end` after the call. Ruby provides a way to check whether a code block has been passed or not, so that you can differentiate the semantics of your internal DSL construct depending on whether it contains a block or not. Using code blocks is common in internal DSLs in Ruby. You can see the same technique applied in the step definitions of the Cucumber project (https://cucumber.io/) for behavior-driven development.

Exercise 10.7. Anonymous functions, code blocks, and passing arguments by-name are commonly used to implement control structures in internal DSLs. Use them to build an imperative state-machine DSL. Change our implementation of

[3] https://expressjs.com/, accessed 2022/09
[4] https://scalatra.org/, accessed 2022/09

```
 1 const useStyles = createUseStyles({
 2   myButton: {
 3     color: 'green',
 4     margin: {
 5       top: 5, // jss-default-unit makes this 5px
 6       right: 0,
 7       bottom: 0,
 8       left: '1rem'
 9     },
10     '& span': {
11       // jss-nested applies this to a child span
12       fontWeight: 'bold' // jss-camel-case turns this into 'font-weight'
13     }
14   },
15   myLabel: { fontStyle: 'italic' }
16 })
```

Figure 10.16: *An example cascading style sheet specified in JSS. Code from https:// cssinjs.org, MIT Licensed*

> the shallow DSL to execute actions instead of producing outputs. The `output` keyword should be changed to `action` and it should take a function of type `Unit => Unit` or a `Unit` value by-name (so `=> Unit`). The transition function will not produce any output, but it will return the target state after executing the action. Alternatively, implement this DSL in Ruby, with code blocks for actions.

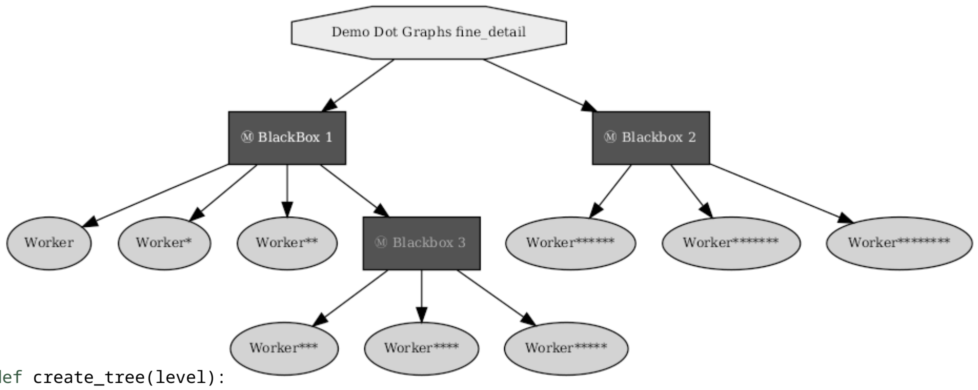
Ruby provides a number of constructs that support meta-programming. You can even overload the dynamic method dispatch mechanism to change how calls are made, for instance to modify the list of parameters before execution. String interpolation, regular expressions, and reflective programming facilities are all used in internal DSLs, which are very popular in the Ruby ecosystem. APIs of many Ruby libraries evolve towards internal DSLs.

JSS: Generating Style Sheets with JavaScript

JSS (Fig. 10.16) is a simple DSL for compact specification of cascading style sheets (CSS).[5] JSS has reached ten million downloads per month from the `npm` repository, with over 700 other packages depending on it. Recall from Chapter 4 that CSS is an external DSL. The JSS language is an internal DSL of JavaScript that allows in-memory dynamic construction of CSS for web applications. We can categorize JSS as deeply embedded—its implementation builds a syntax representation for the CSS, and from then on it is interpreted by the web browser's rendering engine.

The design of the JSS syntax exploits the similarity of the CSS syntax and the syntax of JavaScript object initializers. The entire expression in lines 2–14 of Fig. 10.16 is an object initializer, but it resembles the CSS syntax strikingly (cf. Fig. 3.2, p. 54). The implementation of the DSL performs renaming of attributes (for instance `fontWeight` to `font-weight`, which is not a syntactically correct name for a field in a JavaScript object), applies transformations to sub-trees (l. 10), adds units (not natively supported in JavaScript, see line l. 5), etc.

[5] https://cssinjs.org/, seen 2022/09



```python
def create_tree(level):
    root = py_trees.composites.Selector("Demo Dot Graphs %s" % level)
    first_blackbox = py_trees.composites.Sequence("BlackBox 1")
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.BIG_PICTURE
    second_blackbox = py_trees.composites.Sequence("Blackbox 2")
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.COMPONENT
    third_blackbox = py_trees.composites.Sequence("Blackbox 3")
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.DETAIL
    root.add_child(first_blackbox)
    root.add_child(second_blackbox)
    first_blackbox.add_child(third_blackbox)
    return root
```

Figure 10.17: *Bottom: an internal DSL model of a behavior tree in* `py_trees` *syntax. Top: a visualization of the same model generated using* `py_trees` *tools. Source: https://github.com/splintered-reality/py_trees, BSD Licensed code*

Behavior Trees in Robotics: A Lightweight Deep Embedding

Behavior Trees are a discrete control specification language with similar goals to state machines. Unlike with state machines, in behavior trees one does not define the state transition function explicitly. Instead, a hierarchy of conditional executions is designed. Behavior trees originated in the gaming industry, as a formalism to program autonomous agents in games. They have attracted interest in robotics, where several implementations exist, all as internal DSLs of Python or C++.

The existing implementations of behavior trees use an extremely lightweight form of deep embedding: the API consists of factory functions used to construct an abstract-syntax tree directly. Figure 10.17 illustrates the main idea with an example from the py_trees project.[6] The

[6] https://github.com/splintered-reality/py_trees, retrieved 2022/09

top of the figure shows the visual syntax of a behavior tree, which is essentially also the abstract-syntax tree of the model. This visualization can be obtained using `py_trees` tools. The model itself is not created visually, but programmed in an internal DSL. The code is shown in the bottom of the figure. In Line 2 we create a selector node, also the root of the AST. In lines 3, 8, and 13 we create the black box nodes and connect them to their parent nodes using `add_child`. The remaining lines create the worker nodes and wire the entire tree together using further `add_child` calls. Other implementations of behavior trees in robotics, most notably `BehaviorTrees.CPP`,[7] follow the same pattern. A much larger but essentially similar example in C++ can be seen at https://github.com/kmi-robots/hans-ros-supervisor/blob/master/src/full_supervisor.cpp.

Being a deeply embedded DSL, `py_trees` can offer multiple semantics. The library provides both an interpreter for the models and a visualization tool. Providing these with a shallow embedding would be at least cumbersome, if not impossible. The `BehaviorTree.CPP` library goes further and provides a visual external DSL editor and a concrete syntax based on XML. The tools can generate the C++ code in the internal DSL for execution. Thus `BehaviorTree.CPP` is *both* an internal and an external DSL.[8]

The lightweight deep embedding strategy results in rather hard to read models. Programming behavior trees in these internal DSLs is not much different from using, say, the Ecore API directly to construct class models.[9] On the other hand, the language implementation can be made extremely lean this way. Often this is the easiest way to start an internal DSL, which can later evolve into more complex designs with better concrete syntax.

RxROS: A Dataflow-Oriented Internal DSL for Robotics

RxROS[10] is a DSL for implementing processing nodes in the Robot Operating System (ROS). RxROS is an extension of the popular Reactive Programming paradigm[11] for building ROS systems. Unlike behavior trees, reactive programming is data-flow-oriented, not control-flow-oriented. This is also reflected in the design of the DSL, which allows to construct pipelines or circuits processing events/signals.

Figure 10.18 shows a fragment of an example program in the DSL, and Fig. 10.19 shows the corresponding data-flow graph. The example implements a controller for a remotely operated mobile robot. An operator controls the robot using two redundant interfaces: a joystick and a keyboard. It is instructive to read the code and the diagram in parallel. The stream of events starts with two sources, `/joystick` and `/keyboard`. The two

[7] https://www.behaviortree.dev/, accessed 2022/09

[8] It would be an interesting exercise to attempt to implement a proper internal DSL syntax for behavior trees that would facilitate modular construction of trees and remain readable.

[9] This style was already visible in our transformation rules, for instance in Fig. 7.5 on p. 241 where all the `createXXX` calls are dynamic calls to factories creating abstract-syntax elements.

[10] https://github.com/rosin-project/rxros2, accessed 2022/09

[11] http://reactivex.io/, accessed 2022/09

```cpp
auto joyObsrv = from_topic<teleop_msgs::msg::Joystick>(vpublisher, "/joystick")
    | map([](teleop_msgs::Joystick joy) { return joy.event; });

auto keyObsrv = from_topic<teleop_msgs::msg::Keyboard>(vpublisher, "/keyboard")
    | map([](teleop_msgs::Keyboard key) { return key.event; });

joyObsrv.merge(keyObsrv)
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple)
    | map(velTuple2TwistMsg)
    | sample_with_frequency(frequencyInHz)
    | publish_to_topic<geometry_msgs::Twist>(vpublisher, "/cmd_vel");
rclcpp::spin(vpublisher);
```

Figure 10.18: *A controller for a remotely operated mobile robot in the data-flow style of RxROS, an internal DSL of C++*

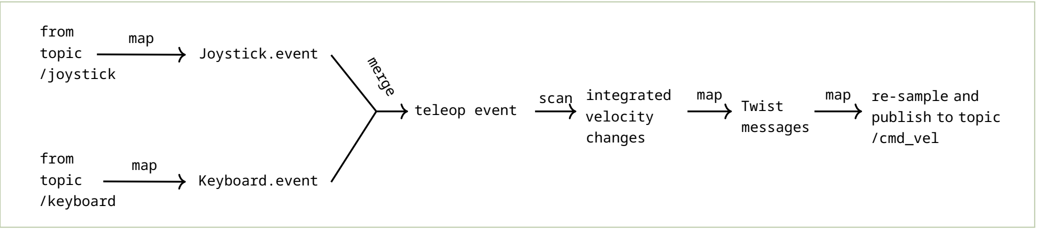


Figure 10.19: *An overview of the data flow in the velocity publisher example from Fig. 10.18*

observers are created in lines 1–2 and 4–5 respectively. In lines 7–11, we construct the flow by merging the streams of events of the two observers, accumulating velocity changes to calculate the desired speed (`scan`), translating the desired velocity into a vector in 3D space (a `Twist`), resampling the stream from the erratic frequency generated by the operator to a fixed frequency required by the motor controller (`sample_with_frequency`) and publishing the resulting stream to the controller (`publish_to_topic`).

RxROS builds on *Reactive Extensions for C++*[12] (RxCPP). Both RxROS and RxCPP are shallowly embedded DSLs in C++. The use of overloaded operators (pipeline) and an expression-oriented data-flow programming create a style that looks alien in C++, especially considering that the classic C++ API is based on callback functions, multi-threading, and locks. The internal DSL program is multi-threaded and safe from deadlocks, but the concurrency and deadlock control are hidden by the abstraction.

Gradle and SBT: Internal DSLs in Build Systems

Internal DSLs in build systems are a long tradition. For example, the KBuild system[13] in the Linux Kernel Project is an internal DSL of GNU Make, and the Colcon system[14] in ROS is an internal DSL of CMake.[15] Figures 10.20 and 10.21 show build scripts for two popular build tools in the JVM ecosystem: Gradle (a popular tool used for compiling, among others,

[12] https://github.com/ReactiveX/RxCpp, accessed 2022/09
[13] https://www.kernel.org/doc/html/latest/kbuild/index.html, accessed 2022/09
[14] https://colcon.readthedocs.io/en/released/user/quick-start.html, accessed 2022/09
[15] https://cmake.org/, accessed 2022/09

```
 1 plugins {
 2   id "base"
 3   id "java"
 4 }
 5 sourceCompatibility = javaVersion
 6 sourceSets { main { java { srcDirs 'src/' } } }
 7 dependencies {
 8   implementation "org.eclipse.emf:org.eclipse.emf.ecore:$emfVersion"
 9   implementation "org.eclipse.emf:org.eclipse.emf.ecore.xmi:$xmiVersion"
10   implementation "org.eclipse.emf:org.eclipse.emf.common:$emfVersion"
11 }
12 tasks.create ('fsmModelCode', GenerateModelCodeTask, 'dsldesign.fsm/model/fsm.genmodel')
13 clean { dependsOn cleanFsmModelCode }
14 compileJava { dependsOn fsmModelCode }
15 jar { from('model') { into('model') } }
```

source: fsm/build.gradle

Figure 10.20: *A fragment of a build script from the book repository, responsible for generating Ecore code for the* `fsm` *meta-model. This is a Gradle build script, simultaneously also a fully functional program in Groovy*

```
name := "option"
scalaVersion := "2.13.3"
scalacOptions += "-deprecation"
scalacOptions ++= Seq ("-deprecation", "-feature", "-Xfatal-warnings")
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.14.0" % "test"
libraryDependencies += "org.scalatest" %% "scalatest-freespec" % "3.2.0" % "test"
libraryDependencies += "org.scalatest" %% "scalatest-shouldmatchers" % "3.2.0" % "test"
libraryDependencies += "org.scalatest" %% "scalatest-mustmatchers" % "3.2.0" % "test"
libraryDependencies += "org.scalatestplus" %% "scalacheck-1-14" % "3.2.0.0" % "test"
```

Figure 10.21: *An example build script for Scala's Simple Build Tool (SBT)*

Android apps) and SBT (the default build tool of the Scala community). Both DSLs are shallowly embedded, in Groovy and Scala respectively.

All keywords in Fig. 10.20 are function calls. Groovy's syntax to call functions without parentheses, and with code blocks in braces, is used heavily. Code blocks are a special case of unary anonymous functions with a default argument, basically delayed computations. For example `plugins` (l. 1) is a function that takes a code block as an argument and executes the code block with an object representing the current configuration as its argument. The configuration object provides a method `id` that registers the needed plugins in the current configuration. The SBT example in Fig. 10.21 does not use functions, but relies on overloading operators (`:=`, `+=`, `++=`, `%%`, and `%`). Both tools allow the user to drop out to the host language for programming complex build tasks, which is commonly done.

Both internal DSLs run on JVM, and this means that they are interoperable across JVM languages. Gradle scripts have historically been designed for Groovy, but nowadays they can also be written in Kotlin. The implementation of Gradle itself mixes Java, Groovy, and Kotlin. Of course, when we change languages the syntax of the internal DSL changes, too. For instance, function calls in Kotlin, unlike in Groovy, require parentheses around arguments (so `id("base")` would be needed in Line 2). On the other

```
enum Camera_States { C_INACTIVE, C_SENSING, C_NOT_SENSING};
SC_MODULE(Camera) {
        sc_in<bool> clk;
        sc_in<bool> rst;
        sc_fifo<int> out_data[1];
        sc_fifo_out<int> out_port[1];
        sc_in<bool> startstop;
        Camera_States currentState;
        int ns, ms;
        PseudoRandom pr;
        void run();
        void sense();
        SC_CTOR(Camera) : ns(0), ms(0) {
                SC_CTHREAD(run, clk.pos());
                reset_signal_is(rst,true);
                currentState = C_INACTIVE;
                for (int i = 0 ; i < 1; i++){
                        out_port[i](out_data[i]);
                }
        }
};
```

Figure 10.22: *A simple SystemC example implementing a state-machine module called "camera" having different inputs/outputs, two empty functions (run, sense) and the constructor SC_CTOR to instantiate the state-machine object. SystemC is an internal DSL of C++ supporting software simulation and high-level hardware synthesis*

```
class Hello extends Module {
  val io = IO (new Bundle {
    val led = Output (UInt (1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U;
  val cntReg = RegInit (0.U(32.W))
  val blkReg = RegInit (0.U(1.W))
  cntReg := cntReg + 1.U
  when (cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Figure 10.23: *A circuit is a class extending the* `Module` *trait provided by the framework. This circuit counts from 0 to 25000000-1, after which it toggles a blinking led and restarts the counter. The assumption is that the circuit is clocked with 50MHz, which gives blinking at 1Hz frequency. Source: Schoeberl [16]*

hand, the same infrastructure, and a very similar DSL, with all the same abstractions, can be reused in or accessed from in many JVM languages.

Internal DSLs are often naturally extensible, including Gradle and SBT. The example in Fig. 10.20 uses a custom task `GenerateModelCodeTask` for code generation. Because this DSL is running on JVM, the extension can be implemented in any JVM language, and we have implemented this new task in Scala, even though it is invoked in a Gradle script—a Groovy program. Internal DSLs are advisable in applications where easy extensibility of the language is a requirement. External DSLs are hard to extend: one needs either to modify their implementation or to implement a built-in extensibility mechanism. The latter comes for free in internal DSLs.

System Design DSLs: SystemC and Chisel

VHDL [7] and Verilog [8] are classic external DSLs for hardware design. Recently the interest in system design languages is shifting towards internal

DSLs, which present a number of advantages, most importantly easy access to good tools, and to test and simulation infrastructure in a software-only environment. The intended users for these DSLs are system engineers who are used to both software development and hardware design. Thus, they are willing to accept the complexity, while being able to appreciate the flexibility.

SystemC (Fig. 10.22) is an increasingly popular language for system design and verification addressing the needs of both software and hardware design. SystemC is an internal DSL of C++, a C++ library that allows one to specify partitioning of the system into components and provide their logic using C++. The choice to make SystemC an internal DSL is key to its success: a system designed in SystemC can be compiled and executed (simulated) in a purely software environment, which facilitates fast development cycles. At the same time, components expressed in a suitable subset can be used for high-level synthesis to create hardware boards using, for instance, FPGA technology. As you can see in the figure, C++ preprocessor macros are used, among other things, to hide C++ syntax and make the domain concepts more directly available to the designer (`SC_MODULE`, `SC_CTOR`, `SC_CTHREAD`) by expanding them to a more complex C++ counterpart. Classes and class templates are used heavily to provide the abstractions.

Using preprocessors is a standard pattern for introducing internal DSLs. The preprocessor translates the new keywords and constructs into the host language—essentially a form of code generation which yields rather simple implementations. Its main disadvantage is that the compiler reports any potential errors at the level of the host, in the generated code. Since no type information is present in simple preprocessors, the programmer has no way to interpret these errors, short of understanding the internals of the implementation of the DSL [2].

Interestingly, SystemC is not the only internal DSL for hardware synthesis. Another interesting language is Chisel[16] (Fig. 10.23). Chisel is a hardware design DSL internal to Scala [16]. The overarching design follows the deep embedding. The first-tier infrastructure generates not an abstract syntax but an intermediate representation FIRRTL, which, like in deeply embedded DSLs, can then be fed to a synthesizer. Chisel programs can be compiled to Verilog. Implicit arguments for function calls and constructors are used to connect the syntax elements with the current execution configuration and the model.

10.4 Guidelines and Techniques for Building Internal DSLs

Selecting the right domain concepts, ensuring static correctness, and providing dynamic semantics are just as essential for internal DSLs as for external ones. Consequently, the syntax, semantics, and domain analysis guidelines from the previous chapters apply to internal DSLs, too. In this section, we complement them with design advice specific to internal DSLs.

[16] https://github.com/chipsalliance/chisel3, retrieved 2022/09

Terminology: Internal DSLs, Embedded DSLs, Embeddings

```
// LINQ Query Syntax (C#)
var result =
  from person in list
  where person.name.Contains("Alice")
  select person.degree;
```

The term of *embedded DSL* is often attributed to Hudak [6] who introduced it in the context of functional programming and Haskell, however the practice was well known in the Lisp community already since the sixties. Hudak's definition of an embedded DSL is essentially the same as our definition for an *internal DSL* (Def. 10.3). Kieburtz [10] calls internal DSLs *open*, emphasizing their extensibility. We prefer the adjective "internal" as it emphasizes that the language is "extracted" from the host by exposing a stylized API. We use the term *embedding* to distinguish the shallow and deep patterns of building internal DSLs.

```
val xmlDoc = <students> // XML Literal in Scala 2
   <person nane="Alice"><points>100</points></person>
   <person name="Bob"><points>315</points></person>
</students>
```

Nowadays, the term *embedded languages* is often used more broadly, referring to languages that are composed into the host language, possibly using additional processing tools. Often such languages cannot be implemented as a library of the host—a program containing DSL fragments might need to be preprocessed, or the host compiler might need to be extended to handle a new language syntax.

Examples of embedded DSLs that are not internal DSLs include *LINQ Query Syntax* in C# (shown in the top of the box), assembly blocks in C/C++ (`asm`), and some of the ways to embed SQL in programming languages. Another example is the embedded XML syntax in Scala (above), which is implemented in the Scala parser, invoking an XML parser. Interestingly, the sentiment in the Scala community is to remove such language extensions from the compiler (see: https://docs.scala-lang.org/scala3/reference/dropped-features/xml.html), migrating the XML support to a proper internal DSL which can be implemented using specialized string interpolators. The tendency to move towards internal DSLs is frequently observed. Language designers equip GPLs with increasingly more powerful extension capabilities, decreasing the need for specialized extensions embedded in the compiler infrastructure while making building internal DSLs easier.

Guideline 10.1 *Build internal DSLs for programmers, not for end-users.* For language engineers, embedding an internal DSL in an expressive host language is the fastest way to obtain a working implementation for many DSLs. For users, an internal DSL is typically far enough from the host language to create an interesting diversion while increasing productivity thanks to high expressiveness; still, close enough not to be a distraction, or to cause technical frictions. Internal DSL programs integrate well into the existing development ecosystem of the host (build systems, debuggers, editors).

As Hudak [6] suggests, abstraction is key to creating high-quality software, and the ultimate abstraction is the language that precisely captures the system's domain. This might be the reason why many software projects and libraries evolve towards internal DSLs, and why, 25 years after Hudak's seminal text, we understand that internal DSLs are a very good tool for efficient programmers, but much less so for the end-users.

From the end-user perspective however, the tool support for internal DSLs tends to be weak: the host language tools are unable to reason at the level of abstraction of the DSL. Syntax highlighting does not highlight the keywords of the DSL, as they are considered identifiers from the host's point of view. On the other hand, the editing and testing environment of the host language *can* be used, and is often sufficiently convenient for programmers. Another good example is error reporting, typically done using the host language infrastructure, or, in other words, not done at all—just left to the type checker of the host language. Users receive messages at a lower level of abstraction than the code they have written in the DSL. More concretely: you cannot get a compile-time message about ambiguity of a grammar from a parser combinator library. You will get an error about type mismatch between some types in the implementation of the parser DSL—often types you have never heard about. While programmers may tolerate this, end-users are immediately confused.

Exercise 10.8. Introduce an error in fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep/coffeeMachine.scala (Fig. 10.1) and compile to see what Scala error is reported. Reflect on what error messages you could expect from an external DSL implementation for a similarly broken input.

Guideline 10.2

Use constants, static values, functions, and methods to introduce keywords. We have demonstrated these mechanisms extensively in the `fsm` language. All methods in Fig. 10.7 introduce keywords. In the Python version, we use several top-level functions, not methods: `initial`, `step`, `transition` (Fig. 10.10). In the shallow DSL, the `state` keyword (Fig. 10.11) is introduced as a global constant, a static value (Fig. 10.12).

When you need to chain keywords, like in "`state machine`," you can combine these techniques: create a unary function `state` that takes an object of a fresh new type "`T`" as an argument, and provide a single global value of that type (`machine: T`). A variation of this scheme provides the `end` keyword in Fig. 10.4. The dual works as well: Create a class "`C`" with a method for the second keyword (`C.machine`) and a global instance of `C` with the name of the first keyword (`state: C`). This is how the `state input` chain is created in Fig. 10.12.

Guideline 10.3

Support optional syntax with overloading, default parameter values, or custom dispatch. The basic way to control syntactic variation is to separate keywords into classes representing distinct parsing contexts. Three classes in Fig. 10.4 demonstrate this pattern: `INITIAL_OR_STATE_OR-_END`, `INPUT_OR_NEXT_STATE`, and `OUTPUT_OR_TARGET`. Each of these offers the keywords available in a particular syntactic location.

In some language designs, a single keyword should offer variation in the same context. For instance, we could imagine a variant of our language that makes the state names optional. This can be done by making the state name parameter take a default value, or by overloading the keyword method in two versions: a variant taking the name and an argument-less variant.

Exercise 10.9. Modify the implementation of the deep `fsm` language (Fig. 10.7) to make the state names optional. Change the abstract syntax to use `Option[String]` instead of `String` for the state name, and adjust the class `INITIAL_OR_STATE_OR`

> `-_END` to assume no state name (`None`) for the state name in `state` and `initial` by default. This exercise can be implemented in Python for Fig. 10.8.

Occasionally it happens that you need more control over the syntax than the type system provides. For instance, default parameter values can typically only be used with the last arguments in the function call. Workarounds for this problem, like named arguments, might destroy the internal DSL illusion of the API. Some object-oriented languages offer a workaround: you can control the method dispatch dynamically at runtime; then you can first inspect the provided arguments, and decide how the execution can proceed. You can even inject new arguments and new functions into classes at runtime, a practice known as *monkey patching*. Injecting new methods at runtime allows the creation of new names in the language based on the previously seen names in the internal DSL model, which could be used to create a form of first-class references to functions or values. For most internal DSLs, such techniques are unjustifiably complex, almost abusive of the host language. They remain useful when there is no other way, at the cost of losing the static control over syntactic correctness of models. Even spelling mistakes can easily be missed by the host infrastructure, and the errors are only reported at runtime. For deeply embedded DSLs, these issues can be partly circumvented by implementing a dedicated static checker invoked before the model is executed.

In Scala, classes that need custom dynamic dispatch must mix in the trait `Dynamic`.[17] and expose new methods by implementing the `applyDynamic` member. The user can then call *any* function name on objects of this class and `applyDynamic` receives all the calls with the function name as a character string, along with a list of arguments. The function can then inspect the arguments and deliver the required behavior. In Python, one can create callable classes to achieve a similar effect (implement `__call__` and dynamically inspect the received arguments). Similar mechanisms exist in all dynamically typed languages, including Smalltalk, Ruby, and JavaScript.

Guideline 10.4 *Integrate with existing types using extension methods.* Monkey patching does allow injection of capabilities of internal DSLs into types that you do not control, for instance to add keywords of the internal DSL to standard library types. Extension methods are a type-safe and easy to use alternative. An extension method is a function added to a class after its implementation but without using inheritance. Extension methods can be called also on objects that are created outside your code, even in code that has been written before the extension methods have been defined.

For a simple example, consider an internal DSL that allows association of English texts with their phonetic encoding, for some linguistic application. In such a language we might want to write:

```
"Hello world" reads "həˈləʊ wɜː(r)ld"
```

[17] https://dotty.epfl.ch/api/scala/Dynamic.html, retrieved 2022/09

If we implement this DSL in Scala, a natural idea is to make `reads` a method of the type `String` which takes another String argument and creates the association in the dictionary of phonetic mappings as a result. Can we add the method `reads` to the `String` class, which is a standard library type? Can we do this so that no other strange problems appear anywhere else in the code that uses `String`? We can define `reads` as an extension method and import it only in the scope of the internal DSL model. For example:

```
extension (s1: String)
  def reads (s2: String) =
    ... // implement the logic that associates s2 with s1
```

Figure 10.24: *An extension method in Scala makes the call* `reads` *available on instances of* `String`

You will find extension methods in most modern GPLs (C#, Scala, Groovy, Kotlin, Xtend). In dynamic languages, resort to monkey patching instead, losing static type safety. In Python, overloading operators is an easy, but unsafe, way to extend—operators can be overloaded for existing types.

Guideline 10.5

Limit access to DSL terms using imports, inheritance, or mixins. Internal DSLs tend to heavily extend the available vocabulary. This can easily cause clashes with other libraries. It is useful to limit the internal DSL API to be available only in the lexical scope of a model definition. One way to do this is to use extension methods, and only import their definitions where a model is being defined (see above). If your DSL models can be encapsulated in a class definition, then another pattern is to import the DSL definition through inheritance or mixing in a trait.

For the example of the state machine in Fig. 10.7 we would encapsulate the entire implementation into a class or a trait. Let's assume that this class is called `FSM`. Then the model definition can be placed in the default constructor of a specializing class. With this pattern the definition of the API provided by `fsm` is not visible anywhere outside the inheriting class, which should limit the risk of clashes with other libraries.

```
class CoffeeMachine extends FSM:
  val m = state machine ... end
```

Figure 10.25: *A model placed in the default constructor of a class (Scala)*

Guideline 10.6

Create visually intuitive syntax with operators. Like for external DSLs, in some situations, especially if there is prior notation to mimic, it may make sense to use operators instead of keywords. In internal DSLs we are restricted to the available possibilities in the host language. Some languages allow definition of new operators (Haskell and Scala), some others only overriding of existing operators (C++, Python). Furthermore, the parsing of the internal DSL expressions will typically be affected by the operator precedence and associativity rules of the host language. (In an external DSL, we can set the associativity and precedence ourselves.) This has affected the choice of operators in Variant IV of the Python example in Figures 10.9 and 10.10, where we wanted to ensure left-associativity. The operators selected (`**`, `%`, `>>`) have decreasing precedence, enforcing left-to-right parsing of each transition.

Guideline 10.7 *Use the host's expression language, variable and function definitions, character strings, regular expressions, dictionary literals, initializers, etc.* Internal DSLs reuse the existing expression language of the host. We can bind expressions to variable names, and can naturally parameterize internal DSL models by placing them in functions and making them dependent on arguments. If new operators, constructors, or combinators are needed we define them as functions, methods, or class constructors, which produce values, so that they can participate in host language expressions. Implementing combinators (keywords) for internal DSLs purely makes the integration with the rest of the host language easier.

Exercise 10.10. Find online information about assertion matchers used in unit-testing frameworks (JUnit's or Scalatest's matchers, or http://hamcrest.org/). Relate them to your knowledge of DSLs. Are matchers as an internal DSL? How are they implemented in the framework that you chose? Is there an explicit abstract-syntax representation? An explicit evaluator?

Guideline 10.8 *Control structures via laziness and call-by-name.* To implement control structures, we typically need a way to embed code that will not be immediately executed, but whose execution can be postponed or skipped. For instance, a branching statement has the following structure:

```
IF CONDITION THEN CODE1 ELSE CODE2
```

To implement it in a shallow DSL, we need to evaluate the condition, but then evaluate only one of `CODE1` and `CODE2`. For a deep DSL, we want to capture the syntax of the condition and both code parts in branches—neither of them gets immediately executed. In modern GPLs we can realize this using nullary lambdas (`() => CODE1`), call-by-name argument passing (Scala, Haskell), or code blocks (Groovy, Ruby, Kotlin). Then a control combinator becomes a higher-order function taking other code as argument.

Table 10.1: Three strategies for implementing control lazily: function closures, call-by-name, and code blocks

IF/THEN/ELSE implementation as a unary function taking as an argument:	
– A nullary function (Scala)	`IF (()=>CONDITION) THEN { ()=>CODE1 } ELSE { ()=>CODE2 }`
– Some code by-name (Scala)	`IF (CONDITION) THEN { CODE1 } ELSE { CODE2 }`
– A code block (Ruby)	`IF { next CONDITION } THEN { CODE1 } ELSE { CODE2 }`

Exercise 10.11. Extend our implementation of the shallow DSL for finite-state machines to execute arbitrary code when a transition is taken, instead of producing a string message. An example instance is shown in Fig. 10.26. The output value type for transitions needs to be eliminated from the implementation, and the output keyword needs to take a unit value by-name and execute it when the output is produced. The same exercise can be implemented for the Python variant of the deep DSL, with lambdas or other callables storing the action in transitions.

Finally, lazy bindings can also be used to create circular structures in the internal DSL as we did in Figures 10.11 and 10.12 to allow cyclic references.

```
lazy val selection: State = (state
  input "tea"     output { print ("serving tea") }  target makingTea
  input "coffee"  output { print ("serving coffee") target makingCoffee
  input "timeout" output { print ("coin returned") }   target initial
  input "break"   output { print ("machine broken!") } target deadlock
)
```

Figure 10.26: *An example instance in a variant of the `fsm` DSL that embeds imperative actions as code, instead of message values. See Exercise 10.11*

```
1  "t0 composed t1 on random input not handled by t1 is the same as t" in {
2    forAll { (inputStr: String) =>
3      forAll { (inputStr1: String) =>
4        whenever (inputStr != inputStr1) {
5          val deadlock = state
6          lazy val t0: State =
7            state.input (inputStr) output (inputStr) target (t0)
8          lazy val t1: State =
9            t0.input (inputStr1) output (inputStr1) target (deadlock)
10         t0.step (inputStr) should be { t1.step (inputStr) }
```

source: fsm.scala/src/test/scala/dsldesign/fsm/scala/internal/shallow/ShallowSpec.scala

Figure 10.27: *A property test for the shallow language of Fig. 10.12. We construct a two-transition state and show that the second transition has no effect on firing the action of the first, over many random inputs. The test uses the scalatest and scalacheck libraries*

Guideline 10.9

Exploit the static type system and type inference for validity checking. Supplement with runtime checks (shallow DSLs) or a standalone custom static checker (deep DSLs). It is important to emphasize that dynamically typed internal DSLs can be implemented in statically typed hosts, and statically typed (deeply embedded) DSLs can be implemented in dynamically typed hosts. As shown in Fig. 10.10, we use types or classes to control the syntax of the internal DSL, even in Python, which is dynamically typed. Type inference has helped us to keep the complex types outside the model in a statically typed host. For shallow DSLs, we can use the static type system of the host (if any) and runtime checks—basically validation of input arguments for each term. For a deeply embedded DSL, we have the option of implementing a standalone type checker that can be invoked at runtime once the AST is constructed, just like for an external DSL.

10.5 Quality Assurance and Testing Internal DSLs

To test an implementation of an internal DSL, use the host's testing frameworks like for any other API testing. The same coverage criteria apply as in previous chapters: you want to test all syntactic elements, all static semantics restrictions, and exercise the implementation of the semantics well.

Figure 10.27 presents a test fragment written in Quickcheck style for the shallow implementation of transitions, testing the semantics of firing transitions. We build a state with two transitions labeled by two different actions (`inputStr` and `inputStr1`) and fire the first of them (Line 10). The test establishes that a state with both transitions behaves the same as a state with just the first one. This test is typical for shallow embeddings, where we cannot access any representations but need to exercise the semantics directly. For deeply embedded DSLs, we can explore the created syntax tree and test properties directly on it, as explored by the following exercise.

Figure 10.28: A negative test for the deep DSL of Fig. 10.7: missing `input`*. The test passes when the example fails to compile*

```
"""val m = (state machine "m1"
               initial "s1" output "coin" target "selection"
            end)""" shouldNot compile
```

source: fsm.scala/src/test/scala/dsldesign/fsm/scala/internal/deep/DeepSpec.scala

Exercise 10.12. In the deeply embedded DSL of Fig. 10.7 we can construct broken machines without the initial state. Write a regression test that demonstrates that this is a problem: build a test case without an initial state and invoke EMF validation to show that the constructed instance violates the meta-model constraints. The validation can be invoked using `validate` from scala/src/main/scala/dsldesign/scala/emf.scala. Either call that function in your test or see how it uses the `EValidator` support from Ecore and invoke it yourself.

We have previously emphasized the importance of using both positive and negative examples for testing. If a syntactic constraint is not tracked by the implementation (as in the above exercise), we can test it using a negative example relatively easily. But what shall we do about properties that are supposed to be tracked by the implementation? Any possible violation will break the compilation (or interpretation) of the host program. The test suite will not compile, rendering automatic testing unusable.

To work around this, we invoke the compiler of the host language from the tests; the test fails if the compiler succeeds on negative examples or if it fails to compile the positive examples. Some testing frameworks do this automatically: they use meta-programming to attempt to compile a piece of code and report a pass when the compilation fails. An example is shown using the Scalatest framework in Fig. 10.28. There, we have a piece of state machine where the keyword `input` and the input label are missing from the state definition. The test passes if the example does not compile, but the test suite continues execution despite the failure.[18]

10.6 Internal DSLs and the Language-Conformance Hierarchy

Recall that instances of an external DSL conform to a grammar defining the language. This is dramatically different for internal DSLs: all instances of internal DSL models always conform to the grammar of the *host language*. Thus a finite-state machine in Figures 10.1, 10.9, and 10.11, whether shallow or deep, primarily conforms to the definition of Scala (respectively Python).

None of the coffee machine models in this chapter conformed to any definition of the external finite-state-machine language. They were programs in the host language. The deeply embedded implementation (Fig. 10.7) produces instances of the finite-state-machine meta-model (Fig. 3.1), but as *outputs*. The DSL models do not conform to this meta-model, but the program they represent *evaluates* to such an instance. Furthermore, the

[18] See https://www.scalatest.org/user_guide/using_matchers#checkingThatCodeDoesNotCompile. Incidentally, the example is written in Scalatest's internal DSL, giving an illusion of English phrases. The matcher `compile` is implemented using macros. It does not call the Scala compiler—the test's outcome is decided while compiling the test suite.

shallow embedding does not produce explicit instances at all! The execution captures the semantics of the model directly, and thus the only conformance we consider is that to the host language definition. The internal DSL definition is represented indirectly in the API, and conformance to the DSL is enforced by type-checking and potential runtime checks.

Further Reading

Outside the LISP tradition [15], the first conscious examples of internal DSLs include languages for multimedia control and robotics applications embedded in Haskell [14, 3]. Today the papers are mostly interesting for historical reasons. They all exploit functional programming for building internal DSLs. While functional programming (chiefly anonymous functions) remains highly relevant in this context, today we know that internal DSLs thrive in most modern programming languages, regardless of the paradigm followed [17].

An early documented case of the deep embedding pattern is shown by Leijen and Meijer [12]. Gibbons and Wu [5] discuss the trade-offs between the deep and shallow embeddings. They demonstrate a fold-based pattern in Haskell that reduces the problem of low separation of concerns in shallow embedding implementations.

Myltsev [13] and the documentation of parboiled2[19] explain the design of this popular parser combinator library for Scala in detail. Chiusano and Bjarnason [1] go even further in their textbook on functional programming in Scala, devoting an entire chapter to a discussion of design decisions in a parser combinator library. They demonstrate that a PEG parser is a functor and monad. Jennings and Beuscher [9] show another hardware description langauge (Verilog), or more precisely its embedded variant in Scheme. Ghzouli et al. [4] present an in-depth analysis of behavior tree languages from a language design perspective. Larsen, Hoorn, and Wąsowski [11] describe the RxROS internal DSLs for reactive processing in robotics in depth, including performance experiments.

Additional Exercises

Exercise 10.13. Implement a deeply embedded DSL to represent numbers in the unary system as shown in the rightmost column of Fig. 10.29. The unary number system is probably the simplest system for representation of information that we can think of. A natural number is represented by the corresponding amount of ones. Design an abstract syntax to represent the models and implement the API to construct representations like the figure shows. Add a method to evaluate models to strings or integers, so that we can print the value of the unary number as a decimal, for example `print(I I I)` should print 3. The figure shows instances in Scala. In other host languages you may need to use the navigation operator, or parentheses around some arguments. If the task turns out to be difficult, consider alternative designs. For instance, require commas, or dots between the digits. Once you have the first design working, it is usually possible to improve it. See also Exercise 10.25.

[19] https://github.com/sirthias/parboiled2/, seen 2022/09

decimal	binary	unary	internal DSL (Exercise 10.13)
1	1	1	`I`
2	10	11	`I I`
3	11	111	`I I I`
4	100	1111	`I I I I`
5	101	11111	`I I I I I`

Figure 10.29: *Decimal numbers, the corresponding binary and unary representations, and examples in an internal DSL*

decimal	internal DSL (Exercise 10.15)
4	`II II`
5	`II III`
6	`III III`
10	`IIIII IIIII`
5	`I III I`

Figure 10.30: *More flexible syntax for the unary numbers DSL*

Exercise 10.14. In continuation of Exercise 10.13, consider a unary representation of the number 4 in your DSL. Insert explicit parentheses and type annotations into the expression representing 4, to reflect how the host language perceives it. See Fig. 10.5 for inspiration. In your design, do all the `"I"` symbols refer to the same host type, value, or function?

Exercise 10.15. Consider the following extension to the unary numbers DSL (Exercise 10.13). We would like to make writing and reading numbers in our syntax easier. Following the telephone number conventions, we would like to glue together pairs, triples, quadruples, and quintuples of digits (cf. Fig. 10.30). Implement this internal DSL. If you have solved Exercise 4.35 on p. 138, then reflect on the differences of addressing the same problem in an internal and external DSL. (This exercise also makes sense for a shallow DSL, an extension of Exercise 10.25).

Exercise 10.16. A binary number only uses digits zero and one, for instance 101 in binary equals $1*2^2+0*2^1+1*2^0=5$. Design a deeply embedded internal DSL for expressions representing binary numbers. Use letters `I` and `O` to represent the digits, so that you avoid clashes with decimal digits of the host language. For example, 101 could be written `I O I`. The expression should construct a representation of the abstract syntax that, when translated (for example with `toString`), will return the character strings representing the decimal number stored.

Exercise 10.17. Ternary numbers, or base-3 numbers, also known as radix-3 numbers, are constructed from digits 0, 1, and 2. We use symbols `O` (as in Opera), `I` (as in Infinity), and Z (as in Zoo) respectively in the internal DSL to avoid conflicts with the regular digits of the host language. So a ternary number 102 (in decimal $1*9+0*3+2=11$) is represented as `I O Z` in our internal DSL. Implement this internal DSL as a deeply embedding API in a host language of your choice.

Exercise 10.18. Design a deeply embedded DSL to represent Roman numerals up to 50. See Fig. 10.31 for a suggestion. This can be implemented either as a deeply or shallowly embedded DSL. If you have solved Exercise 4.31 on p. 137 then compare the solutions and reflect on the differences between the external and internal implementations.

decimal	internal DSL (Exercise 10.18)
4	`I V`
6	`V I`
7	`V I I`
9	`I X`
10	`X`
14	`X I V`

Figure 10.31: A proposal for syntax of Roman numerals as an internal DSL

```
XML {
  tag ("StudyProgram") {
    tag ("name", "value"->"SDT") /
    tag ("course", "name"->"SMDP", "students"->89, "day"->"Mon") {
      tag ("lecture01", "title"->"Introduction") { tag ("cancelled") / }
      tag ("lecture02", "title"->"Algebra") /
    }
    tag ("course", "name"->"SPLC", "students"->21) /
    tag ("course", "name"->"SASP", "students"->10, "day"->"Tue") /
    tag ("course", "name"->"SPLS", "students"->3) /
    tag ("MSc") /
    tag ("fulltime") /
  }
}
```

Figure 10.32: Constructing a simple XML file in an internal DSL

Exercise 10.19. Develop an internal DSL for defining data in XML format. Assume that XML files contain tags and attributes with string or integer values. No free text is allowed inside tags but tags can be nested. Figure 10.32 presents an instance in a hypothetical syntax for inspiration. In the figure, we terminate a tag construction with a slash to indicate that no nested tags are coming. You can design the language so that the terminating slash is not needed, if you dislike it. There exist several internal DSLs for representing XML values, so feel free to search online for further ideas. You might want to use the host's language support for variadic functions (functions with a varying size of argument list). The arrow syntax, used in many languages to create pairs, can be implemented differently. The implementation should allow the constructed document to be serialized into a string representing the XML data. The serializer should not depend on the internal DSL API, only on the representation of the abstract syntax (the meta-model).

Exercise 10.20. Implement an internal DSL for defining formatters for dates stored in simple records. A formatter is an object that given a date converts it to a character string in a user defined format. Figure 10.33 shows an example of a program using the DSL. If you undertake the exercise in Scala, this program should compile and work the same way with your DSL in scope. If you use another host language than Scala, be prepared to modify the syntax to fit the host language. You can choose between a shallow and a deep embedding yourself. The course repository includes an example solution as a shallowly embedded DSL in Scala, but try to design your solution before consulting the repository (dateformatter.scala/).

Exercise 10.21. Reimplement the finite-state-machine DSL in another host than Python and Scala. Decide whether the embedding should be shallow or deep, depending on the available infrastructure, expressiveness, and personal prefer-

```
val d = Date (31, 5, 2022)                             // outputs:
{ yyyy - mm - dd }                          println d // 2022-05-31
{ mm / dd / yyyy }                          println d // 05/31/2022
{ yyyy mm dd }                              println d // 20220531
{ yyyy (".") mm (".") dd }                  println d // 2022.05.31
{ mmm (" ") yy }                            println d // May 22
{ mmm (" ") ddth (", ") yyyy }              println d // May 5th, 2022
{ / mmm (" ") ddth (", ") yyyy / }          println d // /May 5th, 2022/
{ ("(") mmm (" ") ddth (", ") yyyy (")") } println d // (May 5th, 2022)
```

source: dateformatter.scala/src/main/scala/dsldesign/dateformatter/scala/Main.scala

Figure 10.33: An example program using an internal DSL for building date formatters. In all but the first line, a formatter object is constructed in braces. Here, `println` is a method of the formatter class invoked directly to demo. See Exercise 10.20

ences. Discuss the differences between our implementations and yours. Are you using objects or functions to represent the model? What constructs of the host language are you using to obtain the syntax encoding? Do you have better or worse control over static validity rules? Are the error messages for the user more or less readable?

Exercise 10.22. Modify the implementation of the deep finite-state-machine language (Fig. 10.7 or Fig. 10.10) to add a possibility that users provide a documentation string for a machine model. For example, invent and inject your own keyword or a comment operator (say >>) into the design. Alternatively, make the opening or closing keyword already present in the language take an optional comment string as an argument. Multi-line string literals in Scala or Python could be used to represent the comment values. Consider several designs and implement the one you find a good compromise between usability and the implementation cost.

Exercise 10.23. Modify the Python implementation of the finite-state-machine DSL (Fig. 10.10, fsm.py/FsmInternalDeep.py) to enforce that for each transition the output label can only be specified after an input label, and the target is only allowed after input or output but not before. This can be done in at least two ways: either using separate builder classes for different stages like in our Scala design, or by introducing runtime checks (fail if you see an output label before an input). Note that in Python both designs are enforced at runtime unless you use an external type checker.

Exercise 10.24. Reimplement the deeply embedded internal DSL for finite-state machines (Fig. 10.7) so that the final call to `end` is not needed. This requires that every `state` element, every `initial` element, and each transition line produce a valid Ecore model which can be returned directly. Most operations need to be implemented as extension methods on Ecore generated classes. A simpler version of this exercise can be done with your own abstract syntax instead of Ecore.

Exercise 10.25. Implement the unary numbers DSL (see Exercise 10.13) as a shallowly embedded internal DSL. Each sub-expression should produce an integer number directly. You likely need a host language which supports extension methods, either directly or by implicit conversions. After you are done, prepare a fully typed explanation of how the language works in the style of Fig. 10.5.

Exercise 10.26. Generalize the design of the shallow DSL for finite-state machines in Scala (Fig. 10.12) to take any input and output type instead of string for messages. This requires changing the type `State` to be generic, as in `State[ Input, Output]`. Similarly for the other class, `Suspended`. One difficulty with the state type is that we need to provide an error message for the basic transition in line 5. If the output type is generic, the return value can no longer be a string. A cheap solution is to produce `None` instead. Alternatively, make `state` take an error message argument explicitly, or make state require the existence of a type-class, say `ErrMsg[Output]`, and provide the instances for basic types in the DSL library.

Exercise 10.27. Change the shallow implementation of the finite-state-machine DSL to take arbitrary guard conditions on transitions. The simplest way seems to be to accept any function of the type `String => Boolean` that given an input label evaluates a condition to decide whether the transition should fire or be skipped. For example (cf. Fig. 10.11):

```
lazy val selection: State[String] = (state
  input {i => i=="tea" && beans>9}  output "serving tea" target makingTea
```

Exercise 10.28. Implement a shallow variant of the deeply embedded DSL from Exercise 10.16 or Exercise 10.17.

Exercise 10.29. Build an internal DSL for constructing feature models (see Chapter 11, Fig. 11.10). A possible meta-model to use can be found in Figures 3.20 and 3.21. The constructed representation should accept a parameter representing a configuration (for instance a list of selected features, or a map from feature names to Boolean values) and return true if the configuration is an instance of the represented model, false otherwise. First, build a representation without cross-tree constraints. Then consider adding the possibility of expressing cross-tree constraints using the expression language of the host language.

Exercise 10.30. Study Jnario (https://github.com/sebastianbenz/Jnario), an implementation of behavior-driven design in Xtend (or choose any other BDD implementation in your language of choice, say Cucumber or Rspec). The core of Jnario is an internal DSL in Xtend for writing behaviors and specs. Study the implementation of Jnario, and argue why it is an internal DSL. What is its target user base? How is it implemented? Is it deeply or shallowly embedded?

Exercise 10.31. (An extension of Exercise 10.12 on p. 388 for the deeply embedded finite-state-machine language of Fig. 10.7) Write a property-based test that generates random `fsm` models in the internal DSL syntax and checks that all the models randomly constructed validate against the meta-model in Ecore. The implementation of the deep DSL can be found in fsm.scala/src/main/scala/dsldesign/fsm/scala/internal/deep.scala and the test can be added to fsm.scala/src/test/scala/dsldesign/fsm/scala/internal/deep/DeepSpec.scala. This test will fail if your generator can create models without the initial state (expected!). Then refactor the implementation of the deep DSL to pass the test (force that `initial` is required). You can invoke EMF validation using the `validate` function from scala/src/main/scala/dsldesign/scala/emf.scala.

References

[1] Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. Manning, 2014 (cit. p. 389).

[2] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: An annotated bibliography”. In: *SIGPLAN Notices* 35.6 (2000) (cit. p. 381).

[3] Conal Elliott. “An embedded modeling language approach to interactive 3D and multimedia animation”. In: *IEEE Trans. Software Eng.* 25.3 (1999) (cit. p. 389).

[4] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. “Behavior trees in action: A study of robotics applications”. In: *International Conference on Software Language Engineering (SLE)*. Ed. by Ralf Lämmel, Laurence Tratt, and Juan de Lara. ACM, 2020 (cit. p. 389).

[5] Jeremy Gibbons and Nicolas Wu. “Folding domain-specific languages: Deep and shallow embeddings (functional pearl)”. In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Ed. by Johan Jeuring and Manuel M.T. Chakravarty. ACM, 2014 (cit. pp. 364, 371, 389).

[6] Paul Hudak. “Building domain-specific embedded languages”. In: *ACM Comput. Surv.* 28.4es (1996), p. 196 (cit. pp. 357, 382).

[7] IEC/IEEE. *IEC/IEEE International Standard - Behavioural Languages - Part 1-1: VHDL Language Reference Manual*. IEC 61691-1-1:2011(E) IEEE Std 1076-2008. 2011. DOI: 10.1109/IEEESTD.2011.5967868 (cit. p. 380).

[8] IEEE. *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012). 2018. DOI: 10.1109/IEEESTD.2018.8299595 (cit. p. 380).

[9] James Jennings and Eric Beuscher. “Verischemelog: Verilog embedded in Scheme”. In: *DSL*. ACM, 1999 (cit. p. 389).

[10] Richard B. Kieburtz. *Defining and Implementing Closed, Domain-Specific Languages*. Invited talk at the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG). 2000 (cit. p. 382).

[11] Henrik Larsen, Gijs van der Hoorn, and Andrzej Wąsowski. “Reactive programming of robots with RxROS”. In: *Robot Operating System (ROS): The Complete Reference (Volume 6)*. Ed. by Anis Koubaa. Springer, 2021 (cit. p. 389).

[12] Daan Leijen and Erik Meijer. “Domain specific embedded compilers”. In: *DSL*. ACM, 1999 (cit. p. 389).

[13] Alexander A. Myltsev. “Parboiled2: A macro-based approach for effective generators of parsing expressions grammars in Scala”. In: *CoRR* (2019). DOI: https://doi.org/10.48550/arXiv.1907.03436 (cit. p. 389).

[14] John Peterson, Paul Hudak, and Conal Elliott. “Lambda in motion: Controlling robots with Haskell”. In: *Practical Aspects of Declarative Languages (PADL)*. Ed. by Gopal Gupta. Vol. 1551. 1999 (cit. p. 389).

[15] Erik Sandewall. “Programming in an interactive environment: the “LISP” experience”. In: *ACM Comput. Surv.* 10.1 (1978) (cit. p. 389).

[16] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. URL: https://github.com/schoeberl/chisel-book (cit. pp. 380, 381).

[17] Weixin Zhang and Bruno C.d.S. Oliveira. “Shallow EDSLs and object-oriented programming: Beyond simple compositionality”. In: *Art Sci. Eng. Program.* 3.3 (2019) (cit. p. 389).

The species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself.

Leon C. Megginson

11 Software Product Lines

We will now look at the application of MDSE for so-called *software product lines*—portfolios of software variants in a particular application domain. We will discuss the systematic engineering of product lines using methods and tools from the field of software product line engineering (SPLE). This field advocates the creation of configurable software platforms that use MDSE technology. From such platforms, the software products (i.e., the individual variants) can be derived, typically in an automated process supported by interactive configurator tools. As such, software product lines are kinds of software *architectures* that aim at maximizing the reuse of code, the reuse of other software development artifacts, and the reuse of engineering efforts.

In this chapter, our focus is on models and DSLs for the domain *software product lines*. As we will show, real-world product lines typically exhibit large and complex variability that needs to be managed—and effectively managing variability requires modeling it, using dedicated DSLs called *variability-modeling languages*. Intuitively, the software variants that are part of a product line (or that can be derived through configuration from a product line), share commonalities and variabilities—for instance, some functionality is sometimes there, sometimes not. Often, certain functionality also depends on other functionalities. So, these functionalities and their dependencies need to be modeled. To this end, a range of variability-modeling languages has been developed, many of which express the logic that some functionality (referred to as a *feature* in the remainder) can be present or absent in a concrete variant of the product line. However, more expressive languages also exist—for instance, when variants differ in how certain parts are connected with each other, which is called topological variability. For the former kind of variability, using so-called feature- or decision-modeling languages suffices (see our case study on the Linux kernel in Sect. 11.2), while for the latter, dedicated DSLs need to be created (see our case study on fire alarm systems in Sect. 11.5).

11.1 Software Variants

The need for software variants is increasing—not only due to an ever-increasing diversity of hardware, runtime environments, and market segments, but also through new application scenarios for embedded or cyber-physical devices, such as wearables or Internet of Things (IoT) devices. Creating variants of software systems allows organizations to address such varying stakeholder requirements. It allows them to experiment with new

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_11

Feature Models are Languages, Too

Recall that this book is primarily about creating modeling languages, not so much about using languages (but, of course we *use* meta-modeling languages for creating other languages). From the descriptions above, you might think that this chapter is mainly about using languages to model the variability of software product lines. However, the languages that we will discuss are in fact meta-modeling languages with different levels of expressiveness. You will see that, when creating *feature models*, you are actually creating a new language—a specific feature model—that describes the whole product line and that can be instantiated by creating a *configuration*. The latter is a model that represents a concrete software variant and that conforms to the feature model.

As such, this chapter will also introduce you to simpler and less expressive meta-modeling languages than class diagrams. In fact, we will present you a spectrum of languages. Class diagrams are certainly the most expressive ones, and in all the chapters above we have used their power to create DSLs describing possible instances of software systems or parts thereof. But here, you will see that less expressive meta-modeling languages exist that can suffice as well. We will specifically discuss the advantages and disadvantages of using DSLs versus feature models, and also explain the spectrum of languages between the two.

ideas or optimize non-functional requirements, such as performance, power consumption, or cost.

Opportunistic software reuse. Consider a typical scenario of *opportunistic* code reuse, without a product line architecture. In this scenario, a developer clones (copies) a fragment of code that implements some functionality that has already been developed in an existing project. This allows her to reuse past effort very easily and fast, but unfortunately leads to multiplication of maintenance efforts. The cloned code starts to live its own life. If she fixes a bug in it, it is very likely that the bug will persist in the original project. Fixing it there requires additional effort. Also, if the original is fixed, it is unlikely that the correction will be propagated to the new project. Furthermore, all the effort on testing the code is now duplicated in the two projects.

Over time, the software organization will have a number of projects that share pieces of functionality, but that do not really share code. They only contain copies (clones) of similar code. The shared code in a product system decreases, and the product-specific code grows. If this continues, costs will grow with the age of the projects, and ultimately the entire system family may become too expensive to maintain.

Similar problems appear when this so-called *clone & own* [45, 27, 26, 120] reuse is organized using branching in a version-control system. Many developers initially start to use branching or forking to maintain variations of software, but this only works for limited kinds of variations, and even there it is hard to propagate bug fixes between branches and clones. Through parallel development, developers also commonly face merge conflicts [90, 92, 87, 1], which they need to resolve manually. Essentially, clone & own is only manageable if there is just one difference per variant. Then, using the notion of feature branches or using a dedicated branching strategy, such as that of Staples and Hill [121], can help. But, in general, version control is not well suited to organizing many variants of software in parallel over time—representing software evolution in space [12]. It is much better suited to organizing sequential variants—a.k.a. history, representing software evolution in time [12]. Branches and forks should be used to organize the development process (for instance, using feature branches) and not the architecture.

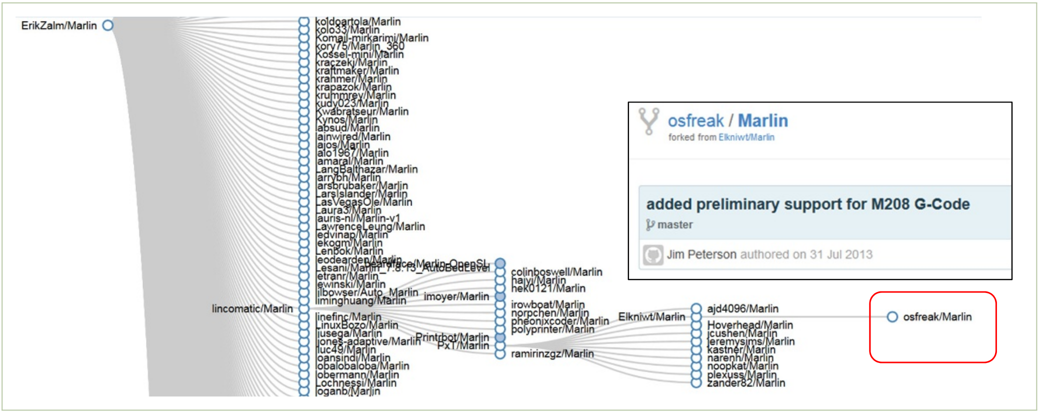
Opportunistic reuse with clone & own is in fact the most common strategy that organizations use for creating software variants [16]. Many companies have documented their experiences [53, 121, 46, 45, 21] of using clone & own. There are also many open-source projects handling their variants with this strategy, such as open-source firmware [120, 82], families of Android apps [26, 94, 26], families of Java and Android games [3, 42, 81], web applications [72], as well as robotics control software [59, 57, 58].

Let us take a look at the 3D printer firmware Marlin [120, 82, 79], which has over 17,000 forks nowadays. Almost 20 % of these forks represent different variants [120], since new features were developed in them, such as to support new printer models. Interestingly, many other forks just change Marlin's configuration file as a pragmatic way to store individual configurations.

Forking provides substantial flexibility and drives innovation in Marlin [120]. It allows experimenting, and the fork developer has full control without affecting the codebase of the main project repository. In fact, forks contributed to the firmware with 58 % of Marlin's commits. With this practice, the Marlin community follows GitHub's recommendation to use forking for developing projects, which is often referred to as pull-based development [60]. In practice, a developer creates a fork, makes modifications, and then creates a pull request to push the changes back to the main project repository, where the changes are reviewed and either merged or rejected. When working on a fork, developers need to pull the recent development changes from the original project repository, which usually evolves when the developer works on the fork.

Marlin also faces the typical problems of clone & own. First, there is the need to propagate changes, especially bug fixes, across the forks. Unfortunately, the propagation of bug fixes is scarce in Marlin. For instance, for a particular bug that crashed the firmware, nine months after it was fixed, only 7 % of the forks had adopted the fix [120]. In general, very few forks (15 %) adopt changes at all. The second typical problem that can be observed with clone & own in Marlin is that it is easy to lose overview of the forks and their content. Finding interesting additions and features becomes challenging. Figure 11.1 shows that sometimes new features can be hidden in the fourth level of forks from the main Marlin project. So, developers can easily lose overview of the features that exist in the fork ecosystem, as well as they may be unaware of the development that is going on.

Marlin is also highly configurable, offering around 140 configuration options in a configuration file to customize Marlin to users' needs and to optimize it with respect to memory consumption. In fact, as software that



osfreak / Marlin
forked from Elkniwt/Marlin

added preliminary support for M208 G-Code
master
Jim Peterson authored on 31 Jul 2013

Figure 11.1: Features hidden in the Marlin fork ecosystem. Source: Ștefan Stănciulescu

runs on embedded systems, the available hardware resources are sparse. As a Marlin contributor acknowledges, "not all boards have enough space to run all the features," which calls for making many of its functionalities (i.e., its *features*) optional. Relying on C++, Marlin adopted the C preprocessor's conditional compilation directives (e.g., `#ifdef`, `#if`) to cut out code from the source files that pertains to disabled features based on the values of the configuration options. Marlin uses these directives to facilitate flexibility of using several variants (e.g., for testing) and to account for memory constraints. Marlin even explicitly prescribes their use to realize optional features and to integrate functionality (features) from a fork in the main project. This way, Marlin benefits from community contributions, while keeping the increase in functionality manageable, still allowing its users to tailor and customize Marlin to their needs.

In summary, Marlin uses clone & own and also some sort of more systematic management of its variability using configuration options and C preprocessor directives. We will now look more into the latter, where we will more abstractly talk about these concepts, referring to configuration options as optional *features* and the preprocessor directives as variation points.

Systematic software reuse. The more variants a system has, the more it needs to adopt dedicated methods and tools to manage variability—or, in other words, to systematically reuse software. Let us look at five large systems that manage vast amounts of variability [15]: the Linux kernel as a general-purpose operating-system kernel, eCos as an operating system for deeply embedded devices, the Debian Linux distribution as a complete operating system with applications, Eclipse as a platform for customizable IDEs with plugins, and Android as a mobile operating system with apps. Each of these has established a software platform with a vibrant software ecosystem around it [24, 71, 15]. In these ecosystems, third-party contributors provide additional value, way beyond what the platform vendors would be able to accomplish. These contributions to the platform have led to vast variability in these five systems. The Linux kernel boasts 15,000 configuration options, allowing it to operate in many different hardware and runtime environments, ranging from Android phones to large supercomputer clusters and server farms. eCos has over 2,800 configuration options and packages to make it run on many different hardware boards. Debian and Eclipse have tens of thousands of software packages and plugins, respectively. Android boasts over 2 million apps today. Each of these software ecosystems uses different variability mechanisms and strategies to systematically manage variability, as illustrated in Fig. 11.2. Linux and eCos use feature models, which are hierarchical menus of configuration options and their dependencies (explained in detail in Sect. 11.4). eCos, Debian, and Eclipse use package-management systems where so-called manifest files describe the variability information (e.g., name and version of a package, dependencies between packages). Eclipse and Android use service-oriented management and execution of apps, which is characterized by dynamic-binding lookup of app dependencies via the capabilities they offer.

Looking at the Linux kernel, eCos, Debian, Eclipse, and Android reveals a spectrum of different variability mechanisms and strategies. In this order, as shown in Fig. 11.2, we can observe that to the left, the domains are highly technical, while those to the right are more end-user-oriented. To the left, we find rather static and closed configuration, where the whole space of configuration options is declared in one model, while to the right, the systems focus more on dynamic and open configuration. The systems to the left also rather strive to control and manage variability, controlling the system's scope, and strictly assuring contribution quality; while those systems to the right focus more on encouraging variability to foster growth of the ecosystem, letting the community decide the scope, encouraging competition and community innovation.

In summary, we can see that using feature models works and scales well for static variability in engineering domains. Feature models support fine-grained, low-level, and controlled configuration. To the contrary, the open and dynamic ecosystems grow fast and, therefore, rely on mechanisms that we call dynamic binding, runtime-service lookup, capability-based dependencies, and easy download and installation. For more details about these mechanisms, we refer to Berger et al. [15].

In the remainder, our focus is on feature models as a language[1] that is not only confined to systems such as the Linux kernel or other software product lines, but feature models can be seen as a very intuitive language to model systems, domains, concepts, or other languages.

11.2 Case Study: The Linux Kernel

Let us look a bit deeper into the Linux kernel's systematic software reuse, specifically its use of variability modeling. Like Marlin, it has tens of thousands of forks, but also systematically reuses software with a highly

[1]In fact, there is no single language, but "feature models" can rather be seen as a family of languages, with a large number of variations proposed in the literature [75, 13].

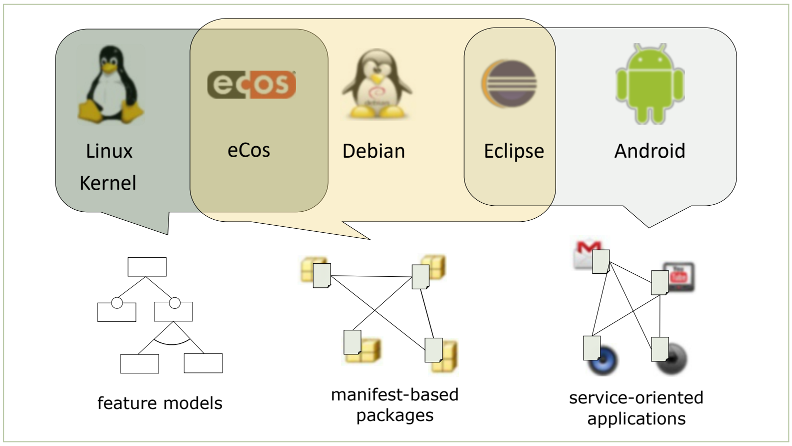


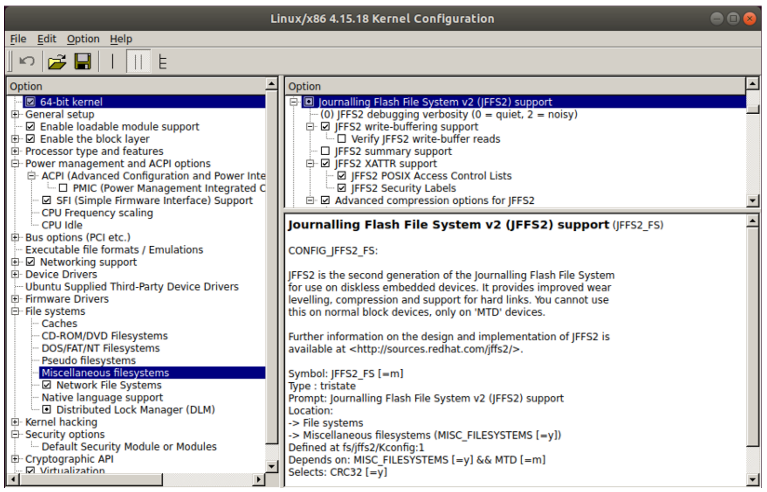
Figure 11.2: *Variability mechanisms in five software ecosystems*

configurable software platform comprising more than 15,000 features today. The majority of these features represents configuration options that can have values of a specific type, most of which control the inclusion of source code for compilation in the build process. The predominant programming language is C. The variability is realized using different mechanisms, including the C preprocessor with its conditional compilation directives (e.g., `#ifdef`), ordinary `if` statements in the C source code, and a configurable build system relying on Make [91]. The former two control the selective compilation of parts of a C file by removing the parts that should not be included for the present configuration, while the build system selectively compiles whole files.

Users configure the kernel interactively via its configurator tool, which exists in three different variants. Figure 11.3 shows a screenshot of the graphical configurator; the other two variants of the configurator are optimized for shell use. While end-users typically do not need to modify the default configuration provided with the Linux distribution that ships the kernel, it is sometimes necessary even for end-users to tweak the kernel towards specific hardware or environments. Linux developers or system integrators modify the configuration much more, allowing the kernel to run in a large range of environments, from supercomputer clusters to Android devices.

Users create a kernel configuration by giving values to features (mainly by selecting or deselecting them) in the configurator tool (see Fig. 11.3). A configuration is an assignment of concrete values to features according to the feature's type and other constraints. To derive a customized Linux kernel, the configuration is then used in the kernel's build process to steer the inclusion of source files [18] for compilation. Specifically, the build system selects the files relevant for the selected features—more precisely, the files whose presence condition (cf. Def. 11.4 below) evaluates to true[2]—and then the C preprocessor outputs C source files that are customized via

[2]It is actually even more complicated, since the build system does not use presence conditions explicitly, but they are encoded using some convention. See Berger et al. [18] for more details.



```
Option
☑ 64-bit kernel
⊞ General setup
  ☑ Enable loadable module support
⊞ ☑ Enable the block layer
⊞ Processor type and features
⊟ Power management and ACPI options
    ⊟ ACPI (Advanced Configuration and Power Inte
        ☐ PMIC (Power Management Integrated C
    ☑ SFI (Simple Firmware Interface) Support
    CPU Frequency scaling
    CPU Idle
⊞ Bus options (PCI etc.)
  Executable file formats / Emulations
⊞ ☑ Networking support
⊞ Device Drivers
  Ubuntu Supplied Third-Party Device Drivers
⊞ Firmware Drivers
⊟ File systems
    Caches
    CD-ROM/DVD Filesystems
    DOS/FAT/NT Filesystems
    Pseudo filesystems
    Miscellaneous filesystems
    ☑ Network File Systems
    Native language support
    ⊡ Distributed Lock Manager (DLM)
⊞ Kernel hacking
⊟ Security options
    Default Security Module or Modules
⊞ Cryptographic API
  ☑ Virtualization
```

```
Option
⊟ ⊡ Journalling Flash File System v2 (JFFS2) support
    (0) JFFS2 debugging verbosity (0 = quiet, 2 = noisy)
  ⊟ ☑ JFFS2 write-buffering support
        ☐ Verify JFFS2 write-buffer reads
    ☐ JFFS2 summary support
  ⊟ ☑ JFFS2 XATTR support
        ☑ JFFS2 POSIX Access Control Lists
        ☑ JFFS2 Security Labels
  ⊟ ☑ Advanced compression options for JFFS2
```

```
Journalling Flash File System v2 (JFFS2) support (JFFS2_FS)

CONFIG_JFFS2_FS:

JFFS2 is the second generation of the Journalling Flash File System
for use on diskless embedded devices. It provides improved wear
levelling, compression and support for hard links. You cannot use
this on normal block devices, only on 'MTD' devices.

Further information on the design and implementation of JFFS2 is
available at <http://sources.redhat.com/jffs2/>.

Symbol: JFFS2_FS [=m]
Type : tristate
Prompt: Journalling Flash File System v2 (JFFS2) support
Location:
-> File systems
-> Miscellaneous filesystems (MISC_FILESYSTEMS [=y])
Defined at fs/jffs2/Kconfig:1
Depends on: MISC_FILESYSTEMS [=y] && MTD [=m]
Selects: CRC32 [=y]
```

Figure 11.3: The Linux kernel configurator, showing the Kconfig language's concrete syntax

conditional compilation directives (e.g., `#ifdef`, `#if`) within these files. The preprocessed source files can then be compiled and linked. In addition to this rather static mechanism (a variation point that is bound at build time cannot be changed without rebuilding the kernel), many features also control so-called loadable kernel modules, which can be loaded dynamically at runtime. With the exception of these modules, very similar mechanisms can be found in many other systems software projects [19] written in C or C++.

Not all combinations of features and their values are valid. A configuration needs to adhere to constraints. Given the sheer size of the kernel, these constraints need to be declared together with the features in a so-called variability model. Constraints mainly arise from dependencies between features [96], for instance, when the code included by one feature references code that is only included in another feature. There are also dependencies between different hardware, which leads to dependencies between device-driver features. Sometimes, developers also declare constraints that prevent combinations of features that have not been tested or are not (yet) supported.

To declare the features together with their constraints and some other meta-information (e.g., feature description), the Linux kernel comes with a DSL called *Kconfig* [131]. The DSL has one graphical and multiple textual syntaxes, implemented in the respective configurator tools (Fig. 11.3 shows the graphical configurator) [54]. The Linux kernel model spans over 1,000 files written in the textual Kconfig syntax and distributed over the kernel codebase, following its structure. To this end, Kconfig offers a simple modularization concept, where (sub-) Kconfig files can be referenced in a Kconfig file and are then included by the configurator. Kconfig and the configurator tool are also used in various other systems software projects, such as Busybox, and embedded libraries, such as uClibc [19].

The most important semantics exhibited by a Kconfig model is called configuration space semantics, meaning that a model describes all possible valid configurations. Another relevant semantics is called ontological semantics, which refers to the hierarchical organization of features. Both semantics are implemented in the configurator tool. For the former, it restricts the valid changes to those that lead to a configuration that still adheres to the constraints. For the latter, the configurator renders a hierarchy of features as a hierarchical menu browseable the users.

The kernel's model and the Kconfig language have evolved continuously since Kconfig was introduced as a DSL in October 2002. As such, both the language and the model are already relatively old, nicely illustrating how such models and languages evolve [85]. We can clearly see that the evolution of the kernel is feature-driven, since the code and the Kconfig model co-evolve. When changing or adding features (e.g., a device driver), developers usually also need to modify Kconfig files or provide a new Kconfig file, respectively.

We say that Kconfig is a feature-model-like language, since its syntax can be mapped to feature models [19, 116, 114]. Feature models are the most popular notation for modeling features and their constraints, and we will discuss them in detail in Sect. 11.4. Like feature models, Kconfig organizes the features in a hierarchy, offers mandatory and optional features, feature groups, and feature types. Using these concepts imposes constraints among features. Any additional constraints (e.g., a dependency between two features, regardless of how far away they are in the hierarchy) can be expressed as so-called cross-tree constraints. To this end, Kconfig provides a simple constraint language with three-state logic [76] for controlling the binding mode of features (a feature of type "tristate" can be set to disable, enable, or compile as module), as well as comparison, arithmetic, and string operators. Furthermore, Kconfig also exhibits concepts that go beyond feature modeling, mainly to scale the model to over 15,000 features. Among others, it offers visibility conditions for features, modularization concepts, default values, and derived features. For a detailed explanation of all these concepts, we refer to a study about the syntax and semantics of Kconfig by Berger et al. [19], as well as a description and extension of the configurator [54]. We will also explain feature modeling in more detail shortly, in Sect. 11.4, and we describe a feature-modeling methodology extensively in Chapter 12.

Let us look at a small excerpt of the Linux kernel model that is shown in Fig. 11.4. It illustrates the definition of features and constraints for an embedded file system included in the kernel that is called Journalling Flash File System (JFFS2). We also show the excerpt with more features in the graphical feature-model syntax in Fig. 11.8. Our model excerpt shows the definition of the following seven features.

- MISC_FILESYSTEMS is a feature that is mainly used to organize the model. Still, it can be selected or deselected, the latter to disable its whole sub-tree comprising many more "miscellaneous" filesystems.

```
menuconfig MISC_FILESYSTEMS
  bool "Miscellaneous filesystems"

  if MISC_FILESYSTEMS

  config JFFS2_FS
   tristate "Journalling Flash File System" if MTD
   select CRC32 if MTD

  config JFFS2_FS_DEBUG
   int "JFFS2 Debug level (0=quiet, 2=noisy)"
   depends on JFFS2_FS
   default 0
   range 0 2
   --- help ---
     Debug verbosity of ...

  config JFFS2_COMPRESS
   bool "Advanced compression options for JFFS2"
   depends on JFFS2_FS

  choice
   prompt "Default compression" if JFFS2_COMPRESS
   default JFFS2_CMODE_PRIORITY
   depends on JFFS2_FS
   config JFFS2_CMODE_NONE
    bool "no compression"
   config JFFS2_CMODE_PRIORITY
    bool "priority"
   config JFFS2_CMODE_SIZE
    bool "size (EXPERIMENTAL)"
  endchoice

endif
```

Figure 11.4: *Kconfig excerpt for a filesystem (JFFS2) available in the Linux kernel, shown in textual concrete syntax*

- JFFS2_FS is the feature that represents the JFFS2 filesystem, which is of type "tristate" and can have three values (similar to Kleene's three-state logic [76]): "y" (yes, compile into the kernel), "n" (no, do not compile at all), or "m" (module, compile the feature as a loadable kernel module). It depends on the two other features MTD and CRC32, each in a slightly different way, but this subtle semantic difference is not so important here. In short, the latter dependency, declared with the keyword `select`, automatically selects the depending feature when a user selects JFFS2_FS, while the former does not (when MTD is disabled, JFFS2_FS is grayed out and cannot be selected).

- The feature JFFS2_FS_DEBUG sets the debugging level as an integer ranging from 0 to 2 (default 0). Notice the keyword `depends on`, which has a dual meaning. It expresses a dependency, but also that the feature should be a sub-feature of JFFS2_FS. Finally, we show the syntax of the feature description in Fig. 11.4, which we, for brevity, omit for the other features in the excerpt.

- The feature JFFS2_COMPRESS enables data compression in the filesystem and is, as a simple configuration option, only a Boolean feature (keyword `bool`). Its parent feature is set to JFFS2_FS, which this option also depends on.
- Thereafter, we see a feature group named "Default compression" with three features in our excerpt of the Linux kernel model. Exactly one of the three features can be selected: JFFS2_CMODE_NONE, JFFS2_CMODE_-PRIORITY or JFFS2_CMODE_SIZE.

Kconfig is a complex DSL with intricate semantics. Consider again the dual meaning of the keyword `depends on`, which can express both a cross-tree constraint and a hierarchy relationship. The latter is not obvious, and there are further ways of (again, rather implicitly) expressing the hierarchy, which illustrates a language design issue. A developer more familiar with curly-brace-dominated languages, such as Java or C, would probably find using parentheses, brackets, or curly braces a more natural way to explicitly represent the feature hierarchy. Many other surprises exist, especially when combining different elements of Kconfig. For instance, default values for features are only defaults when the feature is visible; otherwise, the default determines the feature's value and cannot be changed.

We see various explanations for the complexity of Kconfig:

- First, the configurator tool is not very intelligent, in the sense that it does not support intelligent choice propagation or conflict resolution. A conflict occurs when a user wants to set at least two features to values that violate constraints. The transitivity of dependencies can make the resolution of conflicts challenging. Support for conflict resolution [54] could help users substantially when they need to enable or disable features, which requires enabling or disabling other features, and so on. Presently, Kconfig tries to tackle this problem with imperative choice propagation that is triggered via certain types of constraints (e.g., `select` does choice propagation, but `depends on` does not), which complicates the language and requires the developers who edit the model to already think about choice propagation. Still, despite this mechanism, performing the configuration is still challenging. In fact, a survey among Linux users [67] revealed that it takes 68 % of them a few minutes to activate an inactive feature on average, with 20 % stating even a few dozen minutes. It also revealed that the advice given by the configurator (and feature descriptions) is often incomplete, hard to understand, or incorrect.
- Second, the Kconfig language was not systematically engineered, as opposed to what we advocate in this book. In fact, when Kconfig was introduced in October 2002, the developers decided against another language that came with more intelligent configuration support (based on a reasoner in the background) and a language with simpler syntax and more intuitive semantics. However, Kconfig is a bit more script-like, which generally appeals to Linux developers.

- Third, Kconfig has been continuously extended, together with its configurator tooling. Language evolution is typically required to be backwards compatible, which under long lifespans complicates the language.

Kconfig's complexity makes it challenging to extend the configurator or build further (intelligent) tools to support the Kernel configuration. For instance, it would be valuable to incorporate better choice propagation and intelligent conflict resolution support using off-the-shelf logical reasoners, such as SAT, SMT, or CSP solvers [108]. Using such a reasoner, however, requires the kernel model to be transformed into the logical representation needed by the solver (e.g., a propositional logic formula in conjunctive normal form in the case of a SAT solver). This, in turn, requires the exact syntax and semantics of Kconfig to be understood in order to develop a valid model transformation. Unfortunately, from our own experience [54], reverse-engineering the syntax and semantics of Kconfig is difficult and laborious. Syntax and semantics are hidden in the implementation of the kernel's configurator tool. In our case, we read the Kconfig documentation, tested the behavior of the configurator on small examples, and inspected the configurator's implementation [116, 19]. Formally defining syntax and semantics took over one month, and implementing the transformation into a propositional logic formula another few months. In a more recent effort, we implemented the semantics and a conflict resolution algorithm fully in C [54]. Various other researchers also implemented transformations themselves later, and were also challenged by Kconfig's complexity, as shown in a survey by El-Sharkawy, Krafczyk, and Schmid [114].

Another open-source DSL used in systems software is CDL (Component Definition Language) [19, 17], specifically in the embedded operating system eCos. Compared to Kconfig, CDL has a syntax that is more intuitive for users familiar with curly-brace-like languages and a more obvious semantics, lacking many of the surprising behaviors of Kconfig. The configurator tool for CDL is also more intelligent, as it comes with a built-in reasoner that resolves configuration conflicts automatically, showing users sets of changes that can be made to a configuration to a feature to be set to a certain value.

Exercise 11.1. The Linux kernel and the Android operating system are two prominent examples of variability-rich systems. After we have discussed the Linux kernel's variability in depth, discuss how Android's variability mechanisms differ from those used in the kernel. Discuss the following aspects: the goal of variability, the target users of the products, the representation of variability, the granularity of variability, and two other aspects you find relevant. To learn more about Android's mechanisms, you could read our paper Berger et al. [15].

Note that there is no notion of completeness for your discussion, and obviously, you cannot cover all the details given in the paper. But cover the aspects above (plus the two you find relevant). Imagine you are describing it to a software architect who does not know about either Linux's or Android's variability mechanism, but needs to decide about what mechanisms to use for an architecture she is designing for some software platform.

11.3 Software Product Line Engineering

Let us now look at the SPLE paradigm. Our understanding of the Linux kernel as a highly configurable system will help, since it uses mechanisms known from SPLE. While the Linux kernel originates from practitioners, and SPLE mainly from researchers who worked closely with industry, both came up with similar concepts. However, SPLE is more than just a bunch of mechanisms, it is a paradigm comprising a business, process, architecture, and organizational aspects, providing a tool box and practices for each of these aspects. SPLE gained popularity in the 1990s and early 2000s, but it goes back to research on so-called program families in the 1970s as described by Parnas [100].

SPLE arose from the observation that *opportunistic reuse does not scale with the number of software variants*, as we discussed above. The following is a well-established definition of what a software product line is.

Definition 11.1. *A* software product line *is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*[3]

This definition emphasizes the following core characteristics of software product lines when systematically developed using SPLE. First, a product line represents a *portfolio of software products* ("set of software-intensive systems").[4] Second, SPLE advocates that a software product line is realized via a *configurable (a.k.a. integrated) software platform* ("share a common, managed set") from which the individual variants can be derived, often in an automated and tool-supported process ("in a prescribed way"). Third, SPLE manages the platform using the notion of *feature* ("managed set of features"), which abstractly represent the common and variable functionalities of products in the product line. As such, the individual products, or variants, are defined by the features they provide. Fourth, SPLE is effective when the products pertain to a particular *domain*, which, as you recall from Def. 2.2 in Sect. 2.2, is an area of knowledge containing concepts and terminology understood by practitioners and including the knowledge to build systems in the area [7, 39].

Organizing your software production into a product line is usually linked with an intention to address a certain well-scoped market niche, by providing well-customizable software for this niche and its stakeholders. The production of this software should rely on systematic reuse. As such, the notion of *domain* is crucial. When software systems do not belong to the same domain, it is usually not meaningful to develop them as a product line. They likely do not share enough commonality that can be exploited to establish a platform.

[3] By Northrop [97].

[4] We use the terms variant, product, and system almost synonymously in the remainder.

SPLE is a method in which technical, business, and management issues overlap. Adopting SPLE requires consideration of the four concerns Business, Architecture, Process, and Organization, which is called the BAPO model [127, 98, 84]. The concern *Business* refers to how to generate revenue from the products of a product line. *Architecture* refers to the technical means to build the product line. *Process* refers to the roles, responsibilities, and their relationships in developing the product line and deriving individual products. *Organization* refers to mapping roles (developers and other stakeholders) to organizational structures. An organization needs to consider all these aspects to effectively adopt SPLE. Otherwise, the endeavor of migrating from opportunistic to systematic software reuse is likely doomed to fail.

SPLE Architecture and Variability Mechanisms

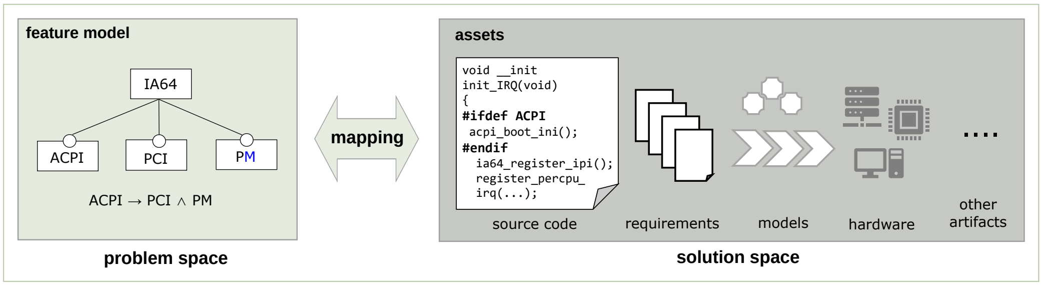
Let us also briefly look at the concern *Architecture*, which typically realizes two abstractions: the problem space and the solution space [39]. Figure 11.6 and Fig. 11.5 illustrate both concepts. The problem space contains the domain-specific abstractions (in our case, features) as an interface to the solution space—the actual software assets in the platform. Both the problem space and the solution space are deep concepts that have been intensively elaborated upon elsewhere [39, 7]. Our focus in this book is the problem space, since we advocate the development of languages providing such domain-specific abstractions. In this chapter, we focus on feature models as a simple and intuitive language to represent the problem space.

For the solution space, you need a product line architecture, which is essentially an MDSE architecture. You apply the same principles, but need to realize variation points that are bound during product derivation. Variation points describe where your system can differ and how, and you realize the using variability mechanisms.

Definition 11.2. *A* variation point *is a specific location in a system where a system can vary in a certain prescribed way.*

Definition 11.3. *A* variability mechanism *is an implementation technique to realize variation points.*

We distinguish between annotative and compositional variability mechanisms. According to Apel et al. [7], the former "annotate a common code base, such that code that belongs to a certain feature is marked accordingly. During product derivation, all code that belongs to deselected features or invalid feature combinations is removed (at compile time) or ignored (at run time) to form the final product." The latter does, according to the same authors "implement features in the form of composable units, ideally one unit per feature. During product derivation, all units of all selected features and valid feature combinations are composed to form the final product." Again, in this chapter, our focus is on the problem space. We refer to the book by Apel et al. [7], which describes many different ways to realize it



```
void __init
init_IRQ(void)
{
#ifdef ACPI
 acpi_boot_ini();
#endif
  ia64_register_ipi();
  register_percpu_
  irq(...);
```

Figure 11.5: High-level architecture of a product line, illustrating problem space, solution space, and the mapping between them

using different implementation techniques. We will, however, discuss those mechanisms in more detail for models instead of code in Chapter 13.

There exists a mapping between problem and solution space, which can be realized using different mapping techniques. An important concept is the presence condition.

Definition 11.4. *A* presence condition *is a logical expression over features determining the presence or absence of software assets in a variant. A presence condition evaluating to true for a specific configuration will include the respective software asset.*

Look at our running example, the Linux kernel, again and observe that it has presence conditions. They are contained in the preprocessor directives (e.g., `#if`) and, implicitly, in its build system [18]. Notably, the presence conditions are not limited to Boolean operators, but also include arithmetic or string operators—essentially the full richness of the C preprocessor.

On a final note, our experience shows that the real benefit of SPLE and feature modeling can only be achieved when the features are mapped to multiple types of assets. For complex and large product lines, mapping to code suffices and already shows the benefits. However, mapping features just to requirements is likely to fail, that is, the costs of doing SPLE and feature modeling exceed the benefits, which arise when new products can be derived quickly in an automated derivation process. Figure 11.5 illustrates a system where features are mapped to code, requirements, models, and pieces of hardware, which is a typical set of asset types that features are mapped to in industrial applications of SPLE.

SPLE Process and Activities

Let us briefly look at the concern *Process*, where SPLE advocates a so-called two-lifecycle process. It separates the development of shared assets (the platform) from the derivation of individual products. Both are full-blown classic engineering processes. Figure 11.6 summarizes these two main (sub-) processes. *Domain engineering* is the process that systematizes and collects knowledge, experience, and assets accumulated in an organization (or in a software project) about a given domain, in order to provide means to reuse

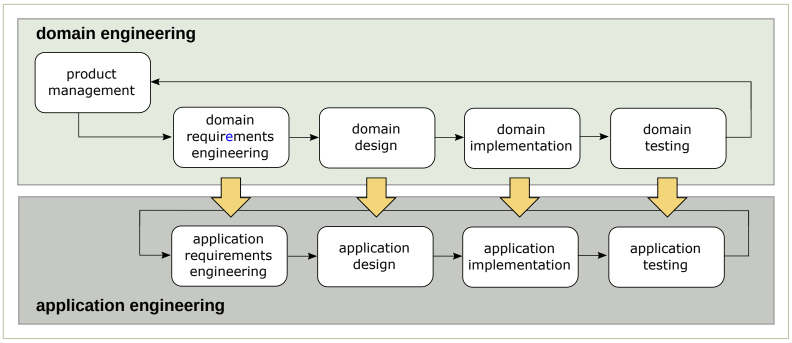


Figure 11.6: The two main processes of SPLE: domain engineering *and* application engineering

these efficiently when building new systems. *Application engineering* (bottom) derives the artifacts from the common domain artifacts produced in domain engineering (the top process). So, the design is done by completion of the shared design, and application development is done by completing/deriving from the framework code. Test cases and documentation might be derived, too. By instituting this process systematically, the cost of obtaining a single product is substantially reduced. Observe that the vertical arrows in Fig. 11.6 (derivation of applications from platform assets) are obtained using technologies presented in the previous chapters. We refer to classical SPLE books for details about the two-lifecycle process [7, 103, 127].

While this two-lifecycle process nicely illustrates the main activities of SPLE and helps in categorizing them, the actual processes in industrial practice usually look different. There is no strict separation between domain engineering and application engineering: instead, activities from both processes are mixed. Organizations often adopt product lines reactively or extractively—they start with one or multiple cloned products and then evolve them into a product line. Furthermore, they often evolve the product line by evolving individual products. Apparently, it is still easier to work on individual products instead of many products (i.e., the platform), but then of course the evolved products need to be integrated into the platform again. In a way, evolution and adoption of product lines then share similar activities. This is reflected in Fig. 11.7, which illustrates adoption (mainly integration of variants) and evolution (mainly evolution of individual products). Read more in Krueger, Mahmood, and Berger [80].

Software Product Lines in Practice

As we have seen, the Linux kernel and other open-source systems software manage their variability relying on techniques known from SPLE and from MDSE, such as a configurable software platform, a configurable build system, a configurator tool, and a DSL- and model-based representation of all features. The latter abstractly represents thousands of variabilities, such as supported drivers, processor architectures, scheduling algorithms, and diagnostic facilities, and the dependencies among them. Even though the Linux kernel was developed completely independently of the research

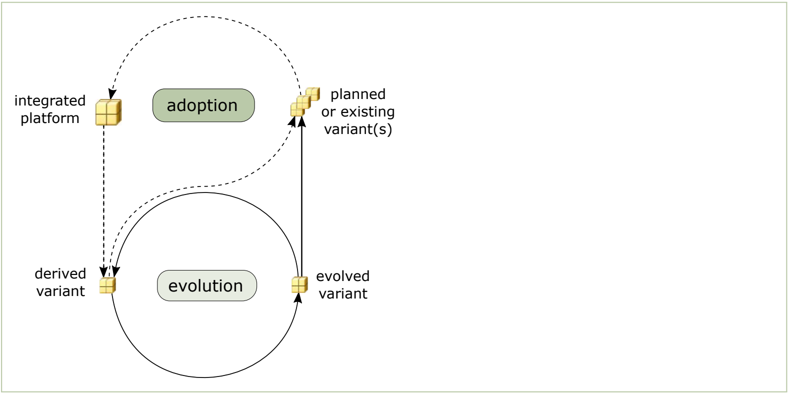


Figure 11.7: Adoption and evolution processes for product lines

community which established SPLE methods and tools since the advent of feature modeling in 1990, it illustrates the practical relevance of SPLE.

Other application domains that typically need to engineer variant-rich systems and that benefit from SPLE are the following.

- The automotive domain boasts some of the largest variant spaces in existence today. SPLE in automotive has been described in experience reports and case studies about Volvo Cars [14] and Scania [49, 61], Audi [64], Daimler [47, 11], General Motors [52], Rolls-Royce [62], as well as the engine control software of Bosch [126] or Cummins [38].
- Avionics and aerospace is another domain benefiting from SPLE, which in addition has strict safety requirements. Example experience reports have been written about Eurocopter [44, 65], Lufthansa [31], NASA [55], Boeing [115], and the US Army's Common Avionics Architecture System (CAAS) [36].
- Telecommunication is another typical domain suited for SPLE. Consider the experience reports about Ericsson [124, 93, 5], E-COM [83], Terrestrial Trunked Radio (TETRA) [102], as well as Nokia Mobile Phones and Nokia Networks [127].
- Power electronics systems often need to exist in many different variants, as discussed in the experience reports about Danfoss [53], ABB [56, 106, 103, 122], and Hitachi [125].
- Robotics and industrial automation systems often come in different hardware configurations and benefit from SPLE methods, as discussed, for example, in a case study on re-engineering automation systems into product lines [77], in experience reports about managing variability in robotics [58, 59], in a case studies on the company Keba [22], and in discussions of systematic variability management in robotics [25, 86].
- Even web applications have been reported [128]. While specific architectures for web applications have been proposed [10], they often exhibit variability in the user interface, which is still difficult to implement [21].
- Further case study collections are provided by van der Linden, Schmid, and Rommes [127], the SEI's catalog of case studies Software Engineer-

ing Institute [118], and the SPLE community's "Hall of Fame."[5] All are summarized in Berger et al. [21].

11.4 Variability Modeling

In this book, we are primarily interested in technical support for software product line engineering. As we have seen in the Linux kernel case study above, MDSE appears very helpful. The idea is to build a variability model of the product line (the Kconfig model in the case of the Linux kernel) that describes the differences and similarities between systems, and then link this model to the implementation either via code generation (generating individual products) or by other means (annotations, preprocessing, interpretation, and so on). Variability modeling is one of the primary means to tackle the complexity of product lines. Such models describe the variability and the commonality of all the variants (i.e., products) that belong to a product line.

Variability modeling can be seen as domain modeling for software product lines or other complex software systems. A variability model is a kind of domain model, since it not only describes the concepts and terminologies in a domain, but also describes what parts of the product line are common to all possible products (or variants) and what are variable (i.e., exist in some, but not all products). Among the latter, there can also be product-specific parts, which only exist in one particular product.

Definition 11.5. A variability model *is a domain model that describes the common and variable aspects of products in a software product line.*

For software product lines, or for any complex system, it is not sufficient to model the variability only—that is, how the individual software products differ. To support the engineering (e.g., to keep an overview understanding or to scope a product line), it is necessary to model the commonality as well.

The first step to build a product line is typically to model the *commonality* and *variability* of the products (i.e., the individual variants) belonging to the product line. Commonality denotes all the aspects that are shared by the products in a software product line. Variability comprises all the aspects in which the products differ. In software product line engineering, the point is to *exploit* the commonality and to *manage* (e.g., limit and scope) variability, in order to obtain faster time to market, and a better return on investment.

Domain

Variability models, and thereby product lines, are usually focused on a specific domain. We distinguish two kinds of domains:

- *vertical domains* are areas which are organized around classes of systems realizing specific business needs, for example "airline reservation systems, order processing systems, inventory management systems" [39].

[5] http://www.splc.net/fame.html

- *horizontal domains* are areas organized around classes of parts of systems (this includes database systems, container libraries, workflow systems, GUI libraries, numerical code libraries, and so on).

One meets product lines in both kinds of domains, but it is most classical to apply SPLE to narrow vertical domains, for example, power electronics firmware or avionics control systems. An example of a product family in a horizontal domain is the Linux kernel,[6] or a configurable platform for cloud computing.

The scope of the domain defines how diverse products will be in this domain. In general, more variability means a wider scope. Remember that variability should be managed, so the scope should be kept under control. The scope of the domain needs to be established based on sales needs, maturity of products and knowledge in the organization, and the potential for reuse. In general, you want the scope be as narrow as possible, and you need to continuously monitor and maintain it, to avoid the *scope-creep* problem. The latter refers to product lines that admit too much variability and become very difficult to maintain (for instance, the products in the product line might no longer share the same core software architecture).

Common and variable properties of the system can be described by a domain model. Such a model defines the scope, the vocabulary, and the main concepts of the domain. Domain models can be expressed in many ways, but are most commonly expressed as a DSL. We call these DSLs variability-modeling languages.

Variability Models

Since the advent of SPLE to efficiently develop software product lines in the early 1990s, a large number of variability-modeling languages have been proposed [117, 33, 4, 32]. Variability modeling is one of the primary means to tackle the complexity of product lines, by describing the variability and the commonality of all the variants that belong to a product line.

The most popular languages are feature [74, 75] and decision models [109], which are relatively similar with only small differences [40]. For instance, the latter's configuration options are called decisions instead of features. Our focus in the remainder is on feature models as the most popular and widespread notation [16]. Furthermore, decision models only focus on variability, without modeling the commonality of product lines, and the notion of feature is more aligned with features as they are commonly used to refer to the functional and non-functional aspects of software systems.

Many other variability-modeling languages beyond feature models exist. We already mentioned *decision modeling*, which originates from the Synthesis method for software reuse [105]. Schmid, Rabiser, and Grünbacher

[6]Some authors would say that highly configurable systems in horizontal domains are not product lines, because they cannot be seen as sets of "products," but rather subsystems or components. Such subtle distinctions are unimportant here, though. They have relatively little influence on the technical aspects interesting for us.

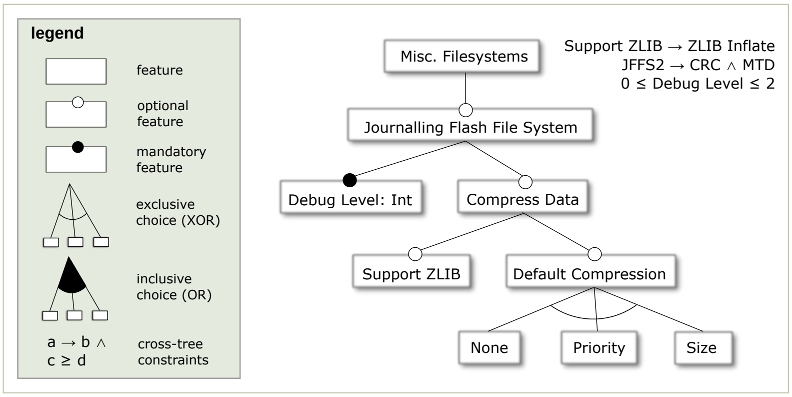


Figure 11.8: Configuration of a Linux filesystem illustrated as a feature model (concrete syntax)

[109], Dhungana, Heymans, and Rabiser [43], and Czarnecki et al. [40] provide further information about decision models. An alternative language is OVM (Orthogonal Variability Modeling), which focuses on modeling variation points and its variants [103]. Surveys comparing variability-modeling languages from different perspectives are provided by Classen, Heymans, and Schobbens [35], Schobbens et al. [110], Schmid, Rabiser, and Grünbacher [109], Czarnecki et al. [40], and Sinnema and Deelstra [117].

An interesting alternative is to provide a variability-modeling language as a UML profile [129, 130], allowing feature models (problem space) and variation points to be added to ordinary UML diagrams, such as class diagrams or state machines. It relies on UML's built-in extension mechanism called profiles [99]. A profile provides so-called stereotypes, which can be added to diagram elements, such as classes, relationships (e.g., associations), or attributes. As such, a standard UML tool can be used for modeling feature models, diagram variation points, and their mappings to features.

Feature Models

Feature models organize features in a hierarchy and declare relationships and constraints among features. Feature models allow developers to keep an overview understanding of software systems, and like features, are an intuitive means for communications, bridging different kinds of stakeholders, including developers and domain or business experts.

Figure 11.8 shows a feature-model example. Take a look at the legend, which explains the basic syntax. *Mandatory* features (filled circle) are features that are always included when their parent is included. *Optional* features (hollow circle) do not need to, but may be included if their parent is included. Both kinds require that their parent is included, though, if they are to be included. *Alternative* feature groups (also called Xor groups) denote an exclusive choice between several alternatives (exactly one needs to be selected with the parent). *Or* group features denote a non-exclusive choice between several alternatives (so more than one inclusion in the group is allowed). Additional dependencies between features (those that cross the tree hierarchy) can be stated on the side.

This description illustrates the so-called configuration space semantics of feature models. Several semantics exist. While the feature hierarchy is one of the most important benefits of feature models (called ontological semantics), allowing engineers to keep an overview understanding of a product line, the primary semantics (called configuration space semantics) of feature models represents the valid combinations and values of features in a concrete product of a product line, restricted by constraints. In other words, the configuration space semantics determines the set of all possible products or variants of a product line. For a detailed description of feature-model semantics, take a look at Sect. 2.3 in the book of Apel et al. [7].

The example model in Fig. 11.8 is a feature model we created for a configurable file system in the Linux kernel called JFFS (Journalling Flash File System). In reality, it is defined in the Kconfig language, which we explained above in Sect. 11.2, and we specifically showed an excerpt of the Kconfig model with the configuration declaration of JFFS2. There, the feature Debug Level is a mandatory feature with the value type integer; Compress Data is an optional feature of type Boolean with the optional sub-features Support ZLIB and Default Compression. The latter is a feature group of type Alternative, allowing exactly one sub-feature to be selected. This example also shows three cross-tree constraints (CTCs), noted next to the diagram. Note that ZLIB Inflate is a feature that is defined outside our excerpt of the Linux kernel model.

Definition 11.6. *A* feature model *is a tree-based structure representing features and their constraints.*

We already mentioned feature models in Chapter 7 and provided a meta-model in Fig. 7.6, and alternative meta-models in Figures 7.7 and 7.9 when we discussed model transformations in Sect. 7.2. For completeness, we include another possible meta-model for feature models in Fig. 11.9. Note that it uses an association class `SubfeatureRelationType` to detail the association between features and their non-grouped (solitary) features, which is not possible in Ecore, but in UML class diagrams. However, in the remainder, it is not essential to understand the meta-models of feature models, since we will use feature models as a meta-modeling language. Feature models are not very expressive, but that is their strength, and convenient tools are available. More complex feature-modeling notations exist. Extensions include adding references between features, and adding classifiers (feature cardinalities, or feature groups).

A product or variant is defined by a configuration of the model.

Definition 11.7. *A* feature-model configuration *is an assignment of concrete values to features. A configuration is an instance of a feature model.*

Mapping features to software assets provides further semantics. The mapping specifies the locations of specific features in the assets. Features can also be mapped to variation points within assets; then they control the inclusion of certain assets depending on the concrete configuration of the feature

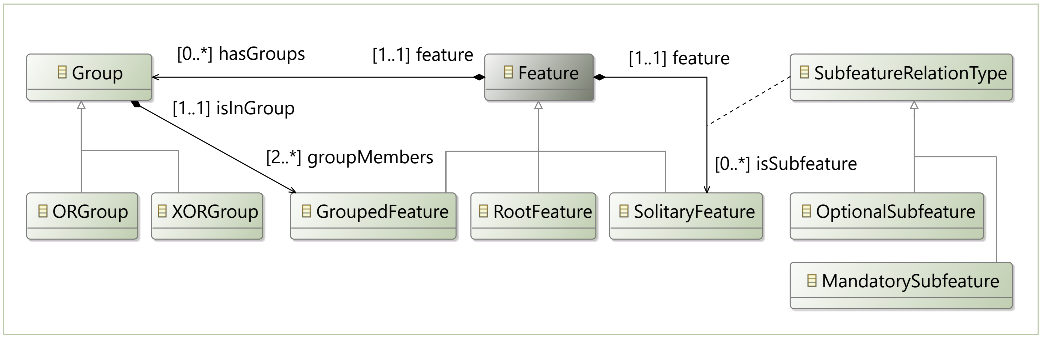


Figure 11.9: One possible meta-model for feature models. Adapted from Janota, Kuzina, and Wąsowski [68]

model. Here, also recall Fig. 11.5, where we discussed the architecture of a product line and how features are mapped to assets. The mapping is exploited in the product derivation process, often performed via a configurable build system, as we described for the Linux kernel in Sect. 11.2 above.

Definition 11.8. Concrete *and* abstract features *are notions referring to the mapping of features to assets. Concrete features are mapped. Abstract features are not mapped and are rather used for model-structuring purposes. They are usually intermediate features in the model hierarchy.*

Exercise 11.2. The company UpAndDown has a competitor, the elevator manufacturer LiftYouUp. One of its customers has an urgent request for an elevator with directed call buttons. Call buttons are either directed call or undirected call. Directed call definitely requires the behavior mode ShortestPath, while undirected call can work with the behavior modes FIFO or ShortestPath. Due to a bug in your current system, ShortestPath does not work with the priority mode RushHourPriority, so you can only sell FloorPriority or PersonPriority currently for ShortestPath (of course, one of these priority modes is required for the elevator to work). FIFO, when used in combination with the priority PersonPriority, excludes undirected call buttons. Overall, you have three available behavior modes, Sabbath, FIFO, and ShortestPath, and all exclude each other. Your customer has heard that some elevators offer periodic airing, which your customer wants, but airing definitely excludes both RushHourPriority and PersonPriority.

Model the problem as a feature model. Can you offer your customer an elevator with directed call buttons and periodic airing?

Exercise 11.3. Draw a feature model for the following product line of (very simple) robot control software.

A robot always has a body, a mobile base, a connectivity system, an arm, and a perception sensor. Optionally, it can incorporate a computer. The mobile base can be biped or wheeled, depending on its operational environment. The connectivity system can be either wireless or wired. If wireless, the connection can be based on Wi-Fi and/or Bluetooth. The end-effector of the robotic arm can be either a parallel gripper (with high payload capacity) or a 5-fingers-hand (provides more functionalities). The perception sensor can be a Lidar and/or an RGBD-camera. The usage of an RGBD-camera requires the inclusion of a computer. If the parallel gripper is chosen, the biped option is not possible.

Exercise 11.4. Draw a feature model for the following subset of the open-source SSL server called AXTLS.

The system supports various platforms, including Linux, Win32, and Cygwin. Exactly one of these platforms has to be selected. AXTLS has a built-in and mandatory HTTP server, which has three optional features: debug mode, HTTP_AUTH authorization, and CGI. The latter is further decomposed into CGI Extensions and LUA scripts which can be enabled for CGI. AXTLS further has so-called BigInt options: an optional sliding window, an optional CRT, and a mandatory reduction algorithm; the latter can be Montgomery, classical or, Barret, or any combination of the three. Montgomery does not work on Cygwin platforms, and Barret requires the debug mode to be enabled.

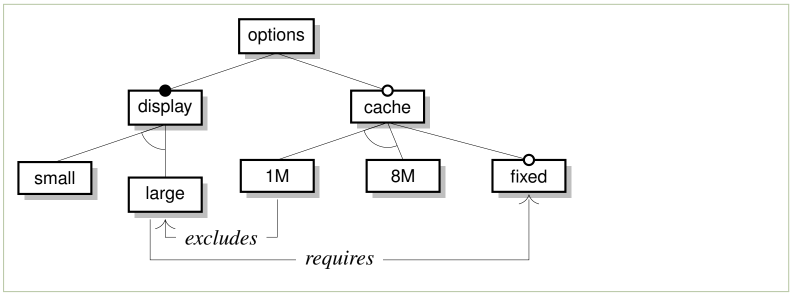


Figure 11.10: A simplified feature model in concrete graphical syntax

Exercise 11.5. Consider the feature model presented in Fig. 11.10. For each of the following configurations state whether it is an instance of the above model:

a) options, display, large, cache, fixed

b) options, display, large, cache, 1M, fixed

c) options, display, small, cache, 8M

d) options, display, small, cache, fixed

Exercise 11.6. Consider the feature model of a car entertainment system presented in Fig. 11.10. Change the model to capture two new requirements:

a) The system should be allowed to have both a small and a large display at the same time (in the above model only one of them is allowed at a time).

b) A system that has *both* a small and a large display, must *also* have an 8M cache.

Recall that you may both modify the diagram and add feature constraints outside the diagram.

Textual Feature Models

Academic feature-modeling languages usually come with a graphical syntax, but there are also textual languages that can be seen as feature-model-like, for instance: TVL [34, 66], Clafer [8, 9], CDL [19, 17], and of course Kconfig, which we discussed extensively above in Sect. 11.2. A comparison of textual languages is provided by Eichelberger and Schmid [48].

Example 27. Let us look at an example of a textual feature-modeling language. Clafer [8] is a language that goes well beyond feature modeling. It allows seamless switching from feature modeling to structural modeling (class modeling). As such, it combines both paradigms (feature and class modeling). Clafer has a very concise syntax, where the feature hierarchy is represented by tab-based indentation. As such, it is a simple and intuitive format for a feature model and can ideally be put into a project to record and organize features without requiring tooling. Such a feature-model file can then be combined with a lightweight annotation system for software artifacts [72, 112] to help record feature locations and to visualize them.

Let us look at an example Clafer model below in Fig. 11.11.

```
1  telematicsSystem
2    xor channel
3      single
4      dual

6    extraDisplay ?
7      xor size
8        small
9        large
10          [ dual  ]
```

Figure 11.11: A simple Clafer model of a car telematics system

A full description of Clafer is provided in Juodisius et al. [73]. In the example, each line has a feature, where the indentation represents the hierarchy. The semantics of the hierarchy is similar to feature models: a sub-feature implies its parent feature. By default, a feature is mandatory (so, the parent also implies the feature), unless made optional by attaching a question mark. When the keyword xor is put directly before the feature name, then its children form an xor group (i.e., exactly one of the children needs to be selected when the feature is selected). Constraints are put in brackets. In our example, large implies that the expression below it in brackets (the indentation expresses that the constraint `[dual]` belongs to the feature large) needs to hold for the feature to be selected. Here, the feature large implies the feature dual. An alternative way of expressing this constraint would be writing `[large => dual]` in line 10, but with the same indentation level as line 7.

Exercise 11.7. Consider the Clafer model from Fig. 11.11 again. For each of the following instances, state whether they adhere or not to the above model.

a) `telematicsSystem`, `channel`, `single`
b) `telematicsSystem`, `channel`, `single`, `extraDisplay`
c) `telemeticsSystem`, `channel`, `single`, `extraDisplay`, `size`, `large`
d) `telemeticsSystem`, `channel`, `single`, `extraDisplay`, `size`, `small`

Exercise 11.8. Consider the Clafer model from Fig. 11.11 again. Change it to capture the following new requirements:

a) The system should be allowed to have both a small and a large extra display at the same time (in the presented model only one of them is allowed at a time).

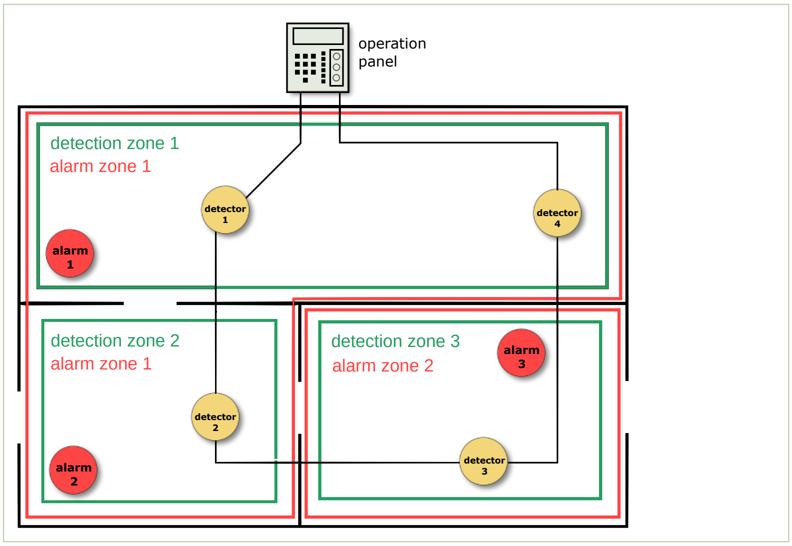


Figure 11.12: *Illustrative example of a simple fire alarm installation [20]*

As in the old model, it is still allowed to have either a small or large extra display alone, and it is still required to have at least a small or large display.

b) If a system has *both* a small and a large display, then it must be dual channel, but a large display should be allowed with a single channel (unlike in the presented model).

11.5 Case Study: A Fire Alarm System

Let us build a meta-model that allows modeling of fire alarm installations. It is based on a real project [20] we conducted with a Norwegian company, Autronica, producing fire alarm systems for industrial plants, oil rigs, and cruise ships. The company used the meta-model for configuring the software controlling the installation of fire alarm devices. While being realistic, the meta-model we will create here is substantially smaller than the real one (which consists of 219 classes). The meta-model represents all possible fire alarm installations the company can deliver, whereas a concrete instance is used to configure the software that runs in special panels (which are usually connected via a network) and controls the installation with all its devices (e.g., smoke detectors or sounders). Figure 11.12 illustrates a simple installation of a fire alarm system.

Motivation. Autronica strives to check rules, regulations, and system constraints at an early stage of the engineering process, well before the delivery starts for each new installation. In the case of fire alarm systems, the configurator not only warrants obtaining the right functionality, but is responsible for enforcing rules required by functional safety certification. Therefore, designing a new AutroSafe installation always involves creating its model. Field equipment is configured by setting various parameters in production and during startup of a panel. In the following, we discuss opportunities and challenges of standardized domain modeling at Autronica.

Modeling configurations using a custom modeling tool. Today, Autronica handles the configuration data systematically and through proprietary configuration tools. The installation configuration model is built by consultants using a custom configurator tool developed around 15 years ago. The tool relies on a meta-model expressed in the Entity-Relationship (E/R) notation. The model has evolved over its lifetime, mainly through additions of new physical devices and relationships. The configurator is used to create one central configuration of the complete installation, which is used to generate C-like data structures for each (display and operation) panel.

Unfortunately, the AutroSafe configuration tool is difficult to maintain, partly because it has been tailor-made and does not rely on any modeling or configuration frameworks. Thus, evolving the tool is a burden. It has served well for years, but the infrastructure provides little overview, and requires complex input. UML modeling tools are much easier to use; they are standardized and maintained. The output from these tools can drive more applications than just configuration, and it is accepted by many other tools thanks to standardization.

Capturing topological properties in domain models. In the legacy E/R model, domain properties are described in a very tight way with a high degree of coupling. Hopefully, using a more developed domain-modeling language will enable a clear separation between the logical and physical topologies, yet still allow constraints relating the two to be described.

Maintaining configurators and meta-models for similar product families. Presently, configurators for several products exist, but they are independently built and rely on different technologies. Some of the input files use XML, others have a C-like syntax. Even though the overall configuration procedures are similar for the products families Autronica does not handle them in a uniform manner.

Abstract syntax (meta-model). A fire alarm installation has a name and a list of responsible persons. The latter are persons who have a name. The installation is composed of multiple domains, which are meant to separate the fire alarm system into parts that should be independent (e.g., when the parts reside in different buildings). For the remainder of the system, the company wanted to be flexible and create a logical structure (for organizing devices into zones) and a physical structure (which reflects the actual, physical layout of the devices on so-called loop cables), so that flexible activation relationships can be realized.

The *logical structure* of fire alarm installations is defined as follows:

- A domain contains one or more operation zones, which have a name, a severity (LOW, MEDIUM, or HIGH), a textual description, and between one and five responsible persons. An operation zone can contain one or more operation zones itself, which makes it possible to divide a zone into sub-zones, allowing an arbitrarily deep hierarchy of zones.
- An operation zone contains an arbitrary number of detection zones and alarm zones, each of which has a name. An alarm zone can have multiple

other alarm zones as neighbors. When an alarm zone starts the alarm, it will notify its neighbors to also trigger the alarm. An alarm zone is mapped to detection zones via an activation expression. This expression can just be the name of a detection zone or a more complex logical expression with the operators AND, OR, and ! (NOT). For instance, if there exist detection zones named D1, D2, D3, D4, D5, one should be able to specify expressions such as:

(D1 AND !D3) OR (D4) OR (D2 AND D4 AND D3)

The *physical structure* is as follows:

- An operation zone is controlled by exactly one panel, which can be a display panel or an operation panel. A panel has a name. An operation panel contains a so-called loop driver module, to which the physical devices are connected (via a loop wire). More precisely, a loop driver contains nodes in a specific order. A node can be either a smoke detector, a sprinkler, or a sounder, each of which has a name. Finally, to connect logical and physical structure, smoke detectors and sprinklers belong to one or multiple detection zones, and sounders belong to one or multiple alarm zones.

Figure 11.13 shows a meta-model realizing the language description above. Note the operation *findDZones* in the class OperationZone, which is a convenience query operation we added to simplify the declaration of a constraint, explained below.

Static semantics (constraints). Let us define the following additional constraints as static semantics in our meta-model.

- Names should be at least two characters long (*invariant nameLength*).
- If the severity of an operation zone is high, then there shall be at least two responsible persons (*invariant: twoResponsibles*).
- Each responsible person shall be responsible for at least one operation zone (*invariant responsibleForOZ*).
- If an alarm zone A is a neighbor of an alarm zone B, then B shall also be a neighbor of alarm zone A (*invariant neighborSymmetry*).
- Alarm zones that are activated by detection zones shall be in the same operation zone (*invariant activatedWithinOZ*).
- An operation zone that is a sub-zone of another operation zone shall be in the same domain as the parent operation zone (*invariant sameDomain*).

We define these constraints using OCL in Fig. 11.14, remembering from Chapter 5 that other constraint languages could be used as well. For descriptions of the OCL language, refer to the sources given in Sect. 5.6, such as the tutorial of Cabot and Gogolla [28]. Note that for the *invariant activatedWithinOZ* we first create a query operation *findDZones* that traverses the expression tree to return the literals (i.e., concrete detection zones) that we then use in the constraint. There, we need to apply this

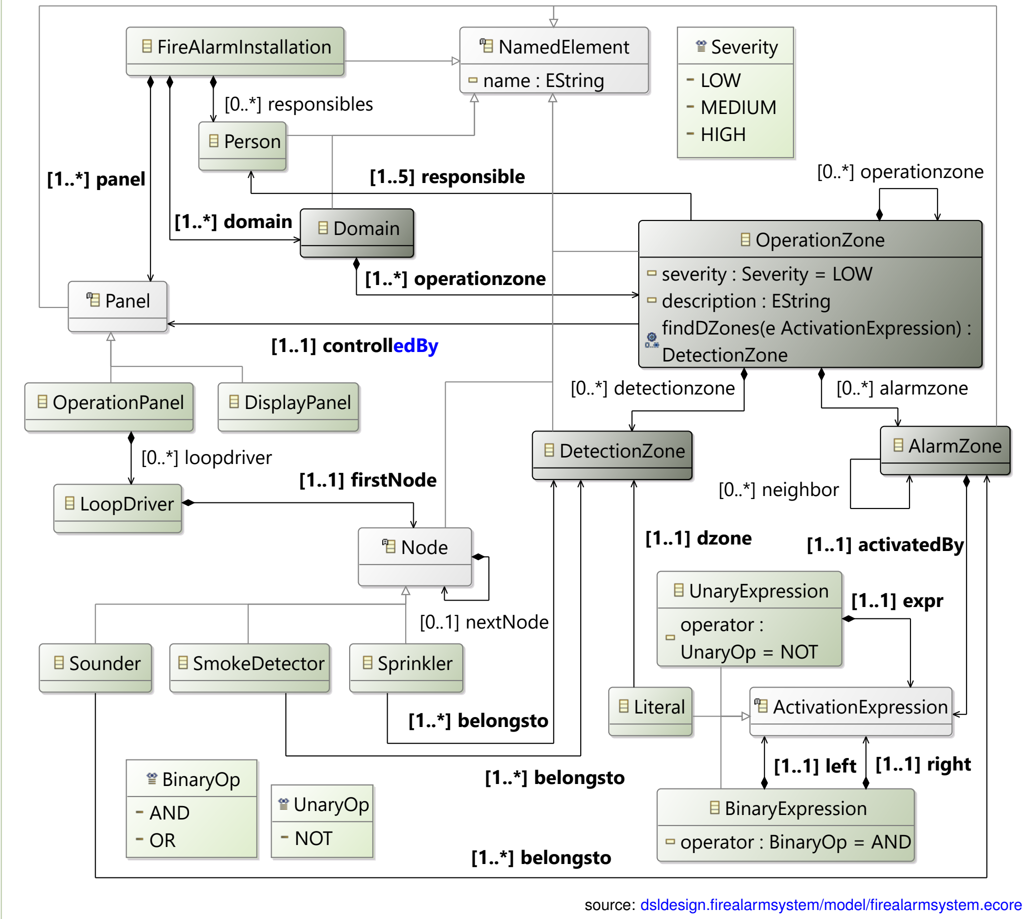


Figure 11.13: *Meta-model for fire alarm installations*

function on a set of activation expressions, which we do via the collection operator iterate. The latter is a common aggregate function in functional programming (e.g., called reduceLeft() in Scala), which aggregates a set via a supplied closure that repeatedly "folds" a set element into an aggregate (which is again a set in our case). Finally, the *invariant sameDomain* is already enforced by the meta-model and can be omitted.

11.6 Spectrum of Variability Modeling

DSLs and feature models are two different techniques that belong to the same continuum of meta-modeling or domain modeling.[7] The spectrum is inspired by Czarnecki and Eisenecker [39] and illustrated in Fig. 11.15. To the left, you go more into routine configuration, which gives you more

[7] See the box on p. 49 for a discussion of the relation of meta-modeling and domain modeling.

```
context NamedElement
invariant nameLength: self.name.size() >= 2

context OperationZone
invariant twoResponsibles: self.severity=Severity::HIGH implies
            responsible->size() >= 2

context Person
invariant responsibleForOZ: OperationZone.allInstances()->
          exists(o : OperationZone | o.responsible->
            exists(p : Person | p = self) )

context AlarmZone
invariant neighborSymmetry: self.neighbor->
            forAll( myNeighbor | myNeighbor.neighbor->
              exists( theirNeighbor | theirNeighbor=self ) )

context OperationZone
findDZones( argument: ActivationExpression ): DetectionZone[*]
body: if argument.oclIsKindOf(BinaryExpression) then
        findDZones(argument.oclAsType(BinaryExpression).left)->
          union(findDZones(argument.oclAsType(BinaryExpression).right))
      else if argument.oclIsKindOf(UnaryExpression) then
        findDZones(argument.oclAsType(UnaryExpression).expr)
      else
        Set{argument.oclAsType(Literal).dzone}
      endif
      endif

invariant activatedWithinOZ: self.detectionzone->includesAll(
            self.alarmzone->iterate( x:AlarmZone; acc=Set{} |
              acc->union( findDZones( x.activatedBy ) ) )
          )

context OperationZone
invariant sameDomain: true    -- already enforced by meta-model
```

source: dsldesign.firealarmsystem/model/firealarmsystem.ecore

Figure 11.14: *Additional constraints (static semantics) for our fire alarm meta-model from Fig. 11.13 defined as OCL constraints*

guidance and efficiency in realizing the problem space. To the right, you go more into creative constructions, which allows more flexibility, but yields more complexity.

Specifically, the spectrum from right to left shows the following variability-modeling strategies:

- If you want to completely avoid variability modeling, then you can write only project-specific code (rightmost) with your own custom code that allows customization of the software to create variants. This is the most flexible and the hardest to maintain mechanism, however. You are not supported by product line or model-driven tooling, and it is up to you whether you want to realize a separation into problem and solution space yourself, which would increase maintainability.
- You can use existing frameworks to realize customization. Then there is at least reuse of the framework, but your framework completion code (what

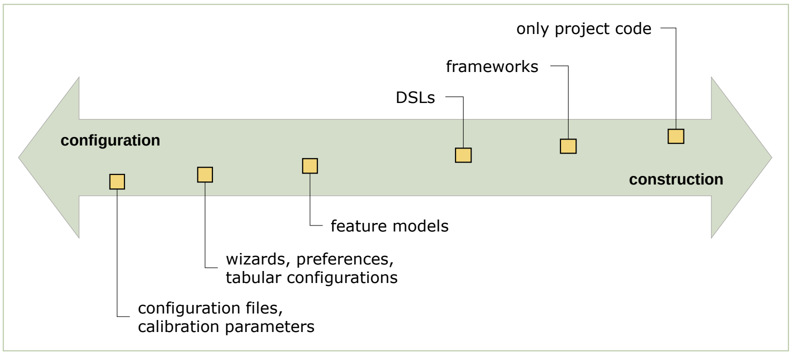

only project code
frameworks
DSLs
configuration
construction
feature models
wizards, preferences, tabular configurations
configuration files, calibration parameters

Figure 11.15: Spectrum of domain modeling, inspired by Czarnecki and Eisenecker [39]

you need to write to use a framework) is still hand-crafted, and it might be difficult to maintain. You will also need to realize separation into problem and solution space yourself in the framework completion code. Frameworks are classified into white-box and black-box frameworks. Read more in Apel et al. [7].

- DSLs and MDSE are closer to the middle of the spectrum. They allow a lot of freedom and flexibility, but are easier to maintain than large amounts of custom code—due to systematic reuse in interpreters and generators, but you need to implement these as well. From here on to the very left you have an explicit separation into problem (e.g., DSL and model) and solution space (e.g., implementation assets).
- Feature models are less expressive than DSLs, and the way to configure with feature models is more rigid than with a typical DSL. Yet, feature models require no meta-modeling and no concrete syntax design, which makes them easier to use. They suffice for many cases.
- Simple configuration files are to the left of the spectrum, and are the cheapest to maintain and the least flexible way to customize a system.

Stahl and Völter [119] advise that one should stay as much as possible to the left side of this spectrum. Namely, one should prefer modeling with feature models, or simple configuration parameters over DSLs, if possible, to avoid scaling up complexity needlessly. The above spectrum makes it clear though, that when features are not sufficient, it is natural to implement product line architectures using DSLs. Consider our case study on fire alarm systems in Sect. 11.5, where feature modeling (where you basically switch on or off features) was not sufficient to represent all the possible concrete fire alarm systems, since that domain is rather about instantiating concepts such as fire detectors and alarm devices and connecting them in a topology. To this end, the expressive power of DSLs was necessary.

Feature Models versus Custom DSLs

Figure 11.16 illustrates the difference between using feature models and creating your own DSL.

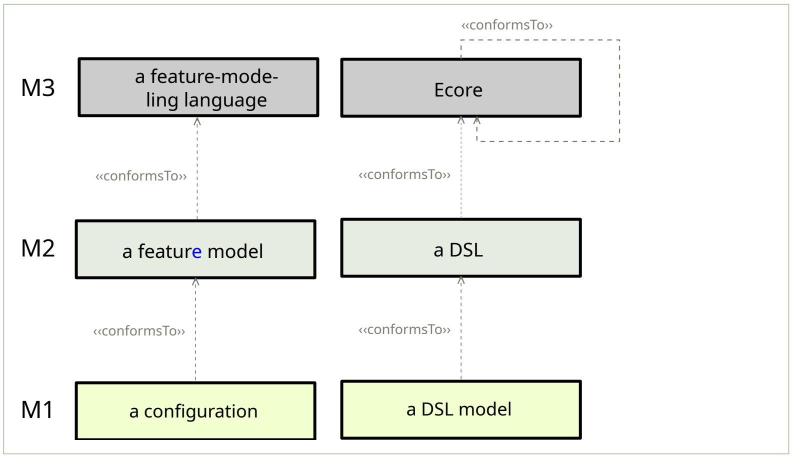


Figure 11.16: Feature models versus custom DSLs

Feature models are *convenient and simple*. There is almost no design effort, and they provide deep insight into the domain as realized in the product line. In contrast, DSLs require more *design and maintenance effort*, but they reward you with more expressiveness. Our fire alarm example (cf. Sect. 11.5) could not have been expressed using feature models. The effort to create feature models is reduced by *existing feature-modeling tools*, while for DSL development, the effort is reduced by *existing language workbenches*.

It is easy to add configuration files to the illustration in Fig. 11.16. At the top (M3), there would be a configuration file schema, such as Java .properties files. At the M2 level, interestingly, one finds the configuration file, which contains both the configuration options and their values. This can be seen as ontological instantiation (cf. Sect. 3.9). So, there is typically no M1 level. Some configuration file mechanisms (not Java .properties files) might allow specification of options and their types, but that is rather uncommon. You would go to feature models to have such support.

Exercise 11.9. Study the way preferences are modeled and realized in Android apps.[a] Extend Fig. 11.16 to show the respective meta-modeling hierarchy for Android app preferences.

Guidelines

Model domain concepts using the following guidelines.

- Concepts that are common to all products in the domain belong to the platform implementation, while concepts and aspects that vary are expressed in your domain-specific models. If you use feature modeling, the mandatory features correspond to common aspects of the system.

[a] See for instance the Android developer guides at https://developer.android.com.

- To decrease complexity, the domain and the platform should be as close to each other as possible. Ideally, the platform (or framework) should provide implementation of domain concepts.
- If you use a DSL, note that, typically, structure is captured in the language, while behavior is provided by the framework/platform. If you do need to customize behavior, it is recommended to reduce it to a small finite number of choices of different behavior and describe it using a feature model.
- If this is not an option, try to reuse as much as possible existing languages such as statecharts, automata, BPMN/BPEL, activity diagrams, and message sequence charts. Designing your own behavioral languages is known to be difficult to get right. Inventing your own behavioral language gives more flexibility than reusing an existing one, but it increases the risks and the difficulty of achieving full automation.
- A good rule of thumb: if you need to introduce typical GPL constructs into your DSL, such as a loop, and they need to be generated from models (compiled into the target language) then you probably have grown your DSL too much. *Most DSLs should stay simple, and possibly declarative.*

In summary, narrow and simple is better than broad and complex.

> **Exercise 11.10.** Discuss the differences between modeling a product line using feature models versus DSLs in a domain of your choice. List at least two advantages of each.

Further Reading

Classical textbooks on SPLE are those by Apel et al. [7], Clements and Northrop [38], Pohl, Böckle, and van der Linden [103], van der Linden, Schmid, and Rommes [127], and Capilla, Bosch, and Kang [29]. Details on implementation techniques for product lines are extensively discussed by Apel et al. [7]. They focus on the solution space and code-level mechanisms, which is a good complement to this and the next chapters, which focus on the problem space and model-based representations of the solution space, as opposed to source code.

Recall the BAPO model, which illustrates four concerns to consider when adopting SPLE. We briefly discussed the concern Architecture in Sect. 11.4 and the concern Process in Sect. 11.3. Read more about poduct line architectures in Balzerani et al. [10] and about dynamic product line architectures in Capilla et al. [30]. Our comparison of variability mechanisms in software ecosystems including Android apps and Debian packages might also be worth reading [15], together with Anvaari and Jansen [6] on the architectural openness of mobile platform architectures and Bosch [23] on architectural challenges in general for software ecosystems.

For product lines, read more about the concern Business in a study on the costs of platform-oriented versus clone & own-oriented reuse in Krueger and Berger [78]. Moreover, Stahl and Völter [119, Chapter 18] state that it often makes economical sense to consider a model-driven product line architecture (PLA) if you can save about 30 % of the code to be maintained. They indicate also that the cost of deriving a new variant is about 20–25 % of making the reference implementation (one variant), and the total saving per early variant is conservatively estimated at 16 %. This gives you some indication of when it makes economical sense to consider SPLE (sometimes people talk about a break-even point at three products for complex systems). The concern Organization is discussed by Clements et al. [37] and Ahmed and Capretz [2]. Furthermore, Fafchamps [50] describes the organizational factors that facilitate or foster software reuse.

To realize product lines, we discussed the clone & own strategy versus the platform strategy in this chapter, and then of course focused on the latter, that is, how it is realized using MDSE techniques. There are various works that focus on bridging the gap between clone & own and platform-orientation, trying to combine the benefits of both. Take a look at frameworks to manage clones by Mahmood et al. [88], Rubin, Czarnecki, and Chechik [107], Fischer et al. [51], Rabiser et al. [104], Martinez et al. [89], Pfofe et al. [101], and Montalvillo and Diaz [95]. A description of development activities supported by automated techniques in the evolution of product lines is given by Strueber et al. [123], who also argue that benchmarks are needed to improve the techniques.

Software ecosystems are conceptual successors of software product lines [24, 70]. While product lines can be seen as perhaps the most successful approach to intra-organizational software reuse, software ecosystems enable inter-organizational reuse. Often, when organizations cannot realize all incoming requirements anymore, they need to open up their platform and allow third-party contributions, essentially establishing an ecosystem in a market niche that strengthens the organization. The challenge of opening up platforms towards software ecosystems has been discussed by Seidl et al. [113], Jansen [69], Schultis, Elsner, and Lohmann [111], Dal Bianco et al. [41], and Hanssen [63], which are all interesting further reads about software ecosystems.

References

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. “Understanding semi-structured merge conflict characteristics in open-source Java projects”. In: *Empirical Software Engineering* 23.4 (2018), pp. 2051–2085 (cit. p. 396).

[2] Faheem Ahmed and Luiz Fernando Capretz. “An organizational maturity model of software product line engineering”. In: *Software Quality Journal* 18.2 (2010), pp. 195–225 (cit. p. 426).

[3] Jonas Akesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. “Migrating the Android Apo-Games into an annotation-based software product line”. In: *SPLC*. 2019 (cit. p. 397).

[4] Vander Alves, Nan Niu, Carina Alves, and George Valen. “Requirements engineering for software product lines: A systematic literature review”. In: *Information and Software Technology* 52.8 (2010), pp. 806–820 (cit. p. 412).

[5] Jesper Andersson and Jan Bosch. “Development and use of dynamic product-line architectures”. In: *IEE Proceedings-Software* 152.1 (2005), pp. 15–28 (cit. p. 410).

[6] Mohsen Anvaari and Slinger Jansen. “Architectural openness: Comparing five mobile platform architectures”. In: *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013, pp. 138–158 (cit. p. 425).

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013 (cit. pp. 406, 407, 409, 414, 423, 425).

[8] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wąsowski. “Feature and meta-models in Clafer: Mixed, specialized, and coupled”. In: *International Conference on Software Language Engineering (SLE)*. 2010 (cit. p. 416).

[9] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. “Clafer: Unifying class and feature modeling”. In: *Software and System Modeling* 15.3 (2016) (cit. p. 416).

[10] Luca Balzerani, D Di Ruscio, Alfonso Pierantonio, and Guglielmo De Angelis. “A product line architecture for web applications”. In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. 2005 (cit. pp. 410, 425).

[11] Joachim Bayer, Thomas Forster, Theresa Lehner, Cord Giese, Arnd Schnieders, and Jens Weiland. “Process family engineering in automotive control systems: A case study”. In: *GPCE*. 2006 (cit. p. 410).

[12] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. “Software evolution in time and space: unifying version and variability management (dagstuhl seminar 19191)”. In: *Dagstuhl Reports*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019 (cit. p. 397).

[13] Thorsten Berger and Philippe Collet. “Usage scenarios for a common feature modeling language”. In: *First International Workshop on Languages for Modelling Variability (MODEVAR)*. 2019 (cit. p. 399).

[14] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. “Three cases of feature-based variability modeling in industry”. In: *MODELS*. 2014 (cit. p. 410).

[15] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. “Variability mechanisms in software ecosystems”. In: *Information and Software Technology* 56.11 (2014), pp. 1520–1535 (cit. pp. 398, 399, 405, 425).

[16] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. “A survey of variability modeling in industrial practice”. In: *VaMoS*. 2013 (cit. pp. 397, 412).

[17] Thorsten Berger and Steven She. *Formal Semantics of the CDL Language*. Tech. note. 2010. URL: https://arxiv.org/abs/2209.11633 (cit. pp. 405, 416).

[18] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wąsowski. “Feature-to-code mapping in two large product lines”. In: *SPLC*. 2010 (cit. pp. 400, 408).

[19] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “A study of variability models and languages in the systems software domain”. In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640 (cit. pp. 401, 402, 405, 416).

[20] Thorsten Berger, Stefan Stanciulescu, Ommund Ogaard, Oystein Haugen, Bo Larsen, and Andrzej Wąsowski. “To connect or not to connect: Experiences from modeling topological variability”. In: *SPLC*. 2014 (cit. p. 418).

[21] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. “The state of adoption and the challenges of systematic variability management in industry”. In: *Empirical Software Engineering* 25 (3 2020), pp. 1755–1797 (cit. pp. 397, 410, 411).

[22] Thorsten Berger et al. “What is a feature? A qualitative study of features in industrial software product lines”. In: *SPLC*. 2015 (cit. p. 410).

[23] Jan Bosch. “Architecture challenges for software ecosystems”. In: *ECSA*. 2010 (cit. p. 425).

[24] Jan Bosch. “From software product lines to software ecosystems”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC. 2009 (cit. pp. 398, 426).

[25] Davide Brugali. “Software product line engineering for robotics”. In: *Software Engineering for Robotics* (2021), pp. 1–28 (cit. p. 410).

[26] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. “Clone-based variability management in the Android ecosystem”. In: *ICSME*. 2018 (cit. pp. 396, 397).

[27] John Businge, Openja Moses, Sarah Nadi, and Thorsten Berger. “Reuse and maintenance practices among divergent forks in three software ecosystems”. In: *Empirical Software Engineering* 27.2 (2022), p. 54 (cit. p. 396).

[28] Jordi Cabot and Martin Gogolla. “Object constraint language (OCL): A definitive guide”. In: *12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-Driven Engineering*. SFM. 2012 (cit. p. 420).

[29] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer, 2013 (cit. p. 425).

[30] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. “An overview of dynamic software product line architectures and techniques: Observations from research and industry”. In: *Journal of Systems and Software* 91 (2014), pp. 3–23 (cit. p. 425).

[31] Gary Chastek, Patrick Donohoe, John D. McGregor, and Dirk Muthig. “Engineering a production method for a software product line”. In: *Proceedings of the 2011 15th International Software Product Line Conference*. SPLC. 2011 (cit. p. 410).

[32] Lianping Chen and Muhammad Ali Babar. “A survey of scalability aspects of variability modeling approaches”. In: *Workshop on Scalable Modeling Techniques for Software Product Lines at SPLC*. 2009 (cit. p. 412).

[33] Lianping Chen, Muhammad Ali Babar, and Nour Ali. “Variability management in software product lines: A systematic review”. In: *SPLC’09*. 2009 (cit. p. 412).

[34] Andreas Classen, Quentin Boucher, and Patrick Heymans. “A text-based approach to feature modelling: Syntax and semantics of TVL”. In: *Science of Computer Programming* 76.12 (2011), pp. 1130–1143 (cit. p. 416).

[35] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. “What’s in a feature: A requirements engineering perspective”. In: *FASE*. 2008 (cit. p. 413).

[36] Paul Clements and John Bergey. *The US Army's Common Avionics Architecture System (CAAS) product line: A case study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 2005 (cit. p. 410).

[37] Paul Clements, Lawrence Jones, Linda Northrop, and John D. McGregor. "Project management in a software product line organization". In: *IEEE Software* 22.5 (2005), pp. 54–62 (cit. p. 426).

[38] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001 (cit. pp. 410, 425).

[39] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000 (cit. pp. 406, 407, 411, 421, 423).

[40] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. "Cool features and tough decisions: A comparison of variability modeling approaches". In: *VaMoS*. 2012 (cit. pp. 412, 413).

[41] Vittorio Dal Bianco, Varvana Myllarniemi, Marko Komssi, and Mikko Raatikainen. "The role of platform boundary resources in software ecosystems: A case study". In: *WICSA*. 2014 (cit. p. 426).

[42] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. "Migrating the Java-based Apo-Games into a composition-based software product line". In: *SPLC*. 2019 (cit. p. 397).

[43] Deepak Dhungana, Patrick Heymans, and Rick Rabiser. "A formal semantics for decision-oriented variability modeling with DOPLER". In: *VaMoS*. 2010 (cit. p. 413).

[44] Frank Dordowsky and Walter Hipp. "Adopting software product line principles to manage software variants in a complex avionics system". In: SPLC. 2009 (cit. p. 410).

[45] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. "An exploratory study of cloning in industrial software product lines". In: *CSMR*. 2013 (cit. pp. 396, 397).

[46] Anh Nguyen Duc, Audris Mockus, Randy Hackbarth, and John Palframan. "Forking and coordination in multi-platform development: A case study". In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2014 (cit. p. 397).

[47] Christian Dziobek, Joachim Loew, Wojciech Przystas, and Jens Weiland. "Functional variants handling in Simulink models". In: *MACDE*. 2008 (cit. p. 410).

[48] Holger Eichelberger and Klaus Schmid. "A systematic analysis of textual variability modeling languages". In: *Proceedings of the 17th International Software Product Line Conference*. 2013 (cit. p. 416).

[49] Ulrik Eklund and Håkan Gustavsson. "Architecting automotive product lines: Industrial practice". In: *Science of Computer Programming* 78.12 (2013), pp. 2347–2359 (cit. p. 410).

[50] Danielle Fafchamps. "Organizational factors and reuse". In: *IEEE Software* 11.5 (1994), pp. 31–41 (cit. p. 426).

[51] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "Enhancing clone-and-own with systematic reuse for developing software variants". In: *30th IEEE International Conference on Software Maintenance and Evolution*. 2014 (cit. p. 426).

[52] Rick Flores, Charles Krueger, and Paul Clements. “Mega-scale product line engineering at General Motors”. In: *Proc. SPLC*. 2012 (cit. p. 410).

[53] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. “Ten years of product line engineering at Danfoss: Lessons learned and way ahead”. In: *SPLC*. 2016 (cit. pp. 397, 410).

[54] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. “ConfigFix: Interactive configuration conflict resolution for the Linux kernel”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021 (cit. pp. 401, 402, 404, 405).

[55] Dharmalingam Ganesan, Mikael Lindvall, Chris Ackermann, David McComas, and Maureen Bartholomew. “Verifying architectural design rules of the flight software product line”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC. 2009 (cit. p. 410).

[56] Christopher Ganz and Michael Layes. “Modular turbine control software: A control software architecture for the ABB gas turbine family”. In: *International Workshop on Architectural Reasoning for Embedded Systems*. 1998 (cit. p. 410).

[57] Sergio Garcia, Daniel Strueber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. “Robotics software engineering: A perspective from the service robotics domain”. In: *28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2020 (cit. p. 397).

[58] Sergio Garcia, Daniel Strueber, Davide Brugali, Alessandro Di Fava, Patrizio Pelliccione, and Thorsten Berger. “Software variability in service robotics”. In: *Empirical Software Engineering* (2022) (cit. pp. 397, 410).

[59] Sergio Garcia, Daniel Strueber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. “Variability modeling of service robots: Experiences and challenges”. In: *13th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 2019 (cit. pp. 397, 410).

[60] Georgios Gousios, Martin Pinzger, and Arie van Deursen. “An exploratory study of the pull-based software development model”. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 345–355 (cit. p. 397).

[61] Hakan Gustavsson and Ulrik Eklund. “Architecting automotive product lines: Industrial practice”. In: *SPLC*. 2010 (cit. p. 410).

[62] Ibrahim Habli and Tim Kelly. “Challenges of establishing a software product line for an aerospace engine monitoring system”. In: *11th International Software Product Line Conference*. SPLC. 2007 (cit. p. 410).

[63] Geir K. Hanssen. “A longitudinal case study of an emerging software ecosystem: Implications for practice and theory”. In: *Journal of Systems and Software* 85.7 (July 2012), pp. 1455–1466 (cit. p. 426).

[64] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. “Reuse of software in distributed embedded automotive systems”. In: *4th ACM International Conference on Embedded Software*. EMSOFT. 2004 (cit. p. 410).

[65] Klaus-Dieter Hess and Frank Dordowsky. “Rational ClearCase migration to a complex avionics project—an experience report”. In: *CONQUEST*. 2008 (cit. p. 410).

[66] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. “Evaluating a textual feature modelling language: Four industrial case studies”. In: *Software Language Engineering*. Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 337–356 (cit. p. 416).

[67] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. “A user survey of configuration challenges in Linux and eCos”. In: *VaMoS*. 2012 (cit. p. 404).

[68] Mikolás Janota, Victoria Kuzina, and Andrzej Wąsowski. “Model construction with external constraints: An interactive journey from semantics to syntax”. In: *MoDELS*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science. Springer, 2008, pp. 431–445 (cit. p. 415).

[69] Slinger Jansen. “Opening the ecosystem flood gates: Architecture challenges of opening interfaces within a product portfolio”. In: *ECSA*. 2015 (cit. p. 426).

[70] Slinger Jansen and Michael A Cusumano. “Defining software ecosystems: A survey of software platforms and business network governance”. In: *Software Ecosystems*. Edward Elgar, 2013 (cit. p. 426).

[71] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. “A sense of community: A research agenda for software ecosystems”. In: *31st International Conference on Software Engineering - Companion Volume*. IEEE. 2009 (cit. p. 398).

[72] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. “Maintaining feature traceability with embedded annotations”. In: *SPLC*. 2015 (cit. pp. 397, 417).

[73] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. “Clafer: Lightweight modeling of structure, behaviour, and variability”. In: *Art Sci. Eng. Program.* 3.1 (2019), p. 2 (cit. p. 417).

[74] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 1990 (cit. p. 412).

[75] Kyo Chul Kang. “FODA: Twenty years of perspective on feature models”. In: *SPLC*. Keynote Address. 2009 (cit. pp. 399, 412).

[76] S.C. Kleene. “On notation for ordinal numbers”. In: *The Journal of Symbolic Logic* 3.4 (1938), pp. 150–155 (cit. pp. 402, 403).

[77] Heiko Koziolek, Thomas Goldschmidt, Thijmen de Gooijer, Dominik Domis, Stephan Sehestedt, Thomas Gamer, and Markus Aleksy. “Assessing software product line potential: An exploratory industrial case study”. In: *Empirical Software Engineering* 21.2 (2016), pp. 411–448 (cit. p. 410).

[78] Jacob Krueger and Thorsten Berger. “An empirical analysis of the costs of clone- and platform-oriented software reuse”. In: *28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2020 (cit. p. 425).

[79] Jacob Krueger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. “Towards a better understanding of software fea-

tures and their characteristics: A case study of Marlin". In: *VaMoS*. 2018 (cit. p. 397).

[80] Jacob Krueger, Wardah Mahmood, and Thorsten Berger. "Promote-pl: A round-trip engineering process model for adopting and evolving product lines". In: *24th ACM International Systems and Software Product Line Conference (SPLC)*. 2020 (cit. p. 409).

[81] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. "Apo-Games: A case study for reverse engineering variability from cloned Java variants". In: *22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. 2018 (cit. p. 397).

[82] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. "Where is my feature and what is it about? A case study on recovering feature facets". In: *Journal of Systems and Software* 152 (2019), pp. 239–253 (cit. p. 397).

[83] Liang Liang, Zhiqiang Hu, and Xiangyun Wang. "An open architecture for medical image workstation". In: *Medical Imaging 2005: PACS and Imaging Informatics*. 2005 (cit. p. 410).

[84] Frank van der Linden. "Software product families in Europe: The Esaps & Café projects". In: *IEEE Software* 19.4 (2002), pp. 41–49 (cit. p. 407).

[85] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "Evolution of the Linux kernel variability model". In: *SPLC*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 136–150 (cit. p. 402).

[86] Alex Lotz, Juan F Inglés-Romero, Dennis Stampfer, Matthias Lutz, Cristina Vicente-Chicote, and Christian Schlegel. "Towards a stepwise variability management process for complex systems: A robotics perspective". In: *Artificial Intelligence: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2017, pp. 2411–2430 (cit. p. 410).

[87] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. "Causes of merge conflicts: A case study of ElasticSearch". In: *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. 2020 (cit. p. 396).

[88] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. "Seamless variability management with the Virtual Platform". In: *43rd International Conference on Software Engineering (ICSE)*. 2021 (cit. p. 426).

[89] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "Bottom-up technologies for reuse: Automated extractive adoption of software product lines". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 2017, pp. 67–70 (cit. p. 426).

[90] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. "Software practitioner perspectives on merge conflicts and resolutions". In: *ICSME*. 2017 (cit. p. 396).

[91] Robert Mecklenburg. *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. O'Reilly Media, Inc., 2004 (cit. p. 400).

[92] Gleiph Ghiotto Lima de Menezes. "On the nature of software merge conflicts". PhD thesis. Federal Fluminense University, Dec. 2016 (cit. p. 396).

[93] Parastoo Mohagheghi and Reidar Conradi. "An empirical investigation of software reuse benefits in a large telecom product". In: *ACM Trans. Softw. Eng. Methodol.* 17.3 (June 2008), 13:1–13:31 (cit. p. 410).

[94] Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. "A large scale empirical study on software reuse in mobile apps". In: *IEEE Software* 31.2 (Mar. 2014), pp. 78–86 (cit. p. 397).

[95] Leticia Montalvillo and Oscar Diaz. "Tuning GitHub for SPL development: Branching models & repository operations for product engineers". In: *SPLC*. 2015 (cit. p. 426).

[96] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. "Where do configuration constraints stem from? An extraction approach and an empirical study". In: *IEEE Transactions on Software Engineering* 41.8 (2015), pp. 820–841 (cit. p. 401).

[97] Linda M. Northrop. "Introduction to software product lines". In: *SPLC*. 2010 (cit. p. 406).

[98] H. Obbink, J. Müller, P. America, R. van Ommering, G. Muller, W. van der Sterren, and J.G. Wijnstra. "COPA: A component-oriented platform architecting method for families of software-intensive electronic products". In: *Tutorial for SPLC* (2000) (cit. p. 407).

[99] Object Management Group. *Unified Modeling Language Specification 2.5.1*. https://www.omg.org/spec/UML. 2017 (cit. p. 413).

[100] David Parnas. "On the design and development of program families". In: *IEEE Transactions on Software Engineering* SE-2.1 (Mar. 1976), pp. 1–9 (cit. p. 406).

[101] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. "Synchronizing software variants with VariantSync". In: *Proceedings of the 20th International Systems and Software Product Line Conference*. 2016 (cit. p. 426).

[102] Pietu Pohjalainen. "Bottom-up modeling for a software product line: An experience report on agile modeling of governmental mobile networks". In: *Proceedings of the 15th International Software Product Line Conference*. SPLC. 2011 (cit. p. 410).

[103] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005 (cit. pp. 409, 410, 413, 425).

[104] Daniela Rabiser, Paul Grünbacher, Herbert Prähofer, and Florian Angerer. "A prototype-based approach for managing clones in clone-and-own product lines". In: *20th International Systems and Software Product Line Conference*. 2016 (cit. p. 426).

[105] *Reuse-driven software processes guidebook, Version 02.00.03*. Tech. rep. SPC-92019-CMC. 1993 (cit. p. 412).

[106] Andreas Rösel. "Experiences with the evolution of an application family architecture". In: *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*. 1998 (cit. p. 410).

[107] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Cloned product variants: From ad-hoc to managed software product lines". In: *STTT* 17.5 (2015), pp. 627–646 (cit. p. 426).

[108] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 2016 (cit. p. 405).

[109] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. “A comparison of decision modeling approaches in product lines”. In: *VaMoS*. 2011 (cit. pp. 412, 413).

[110] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. “Feature diagrams: A survey and a formal semantics”. In: *RE*. 2006 (cit. p. 413).

[111] Klaus-Benedikt Schultis, Christoph Elsner, and Daniel Lohmann. “Architecture challenges for internal software ecosystems: A large-scale industry case study”. In: *FSE*. 2014 (cit. p. 426).

[112] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. “A common notation and tool support for embedded feature annotations”. In: *24th ACM International Systems and Software Product Line Conference - Volume B*. 2020 (cit. p. 417).

[113] Christoph Seidl, Thorsten Berger, Christoph Elsner, and Klaus-Benedikt Schultis. “Challenges and solutions for opening small and medium-scale industrial software platforms”. In: *21st International Systems and Software Product Line Conference (SPLC)*. 2017 (cit. p. 426).

[114] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. “Analysing the Kconfig semantics and its analysis tools”. In: *GPCE*. 2015 (cit. pp. 402, 405).

[115] David C. Sharp. “Reducing avionics software cost through component based product line development”. In: *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No. 98CH36267)*. 1998 (cit. p. 410).

[116] Steven She and Thorsten Berger. *Formal Semantics of the Kconfig Language*. Tech. note. 2010. URL: https://arxiv.org/abs/2209.04916 (cit. pp. 402, 405).

[117] Marco Sinnema and Sybren Deelstra. “Classifying variability modeling techniques”. In: *Information and Software Technology* 49.7 (2007), pp. 717–739 (cit. pp. 412, 413).

[118] Software Engineering Institute. *SEI Product Line Bibliography*. URL: http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm (cit. p. 410).

[119] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. Wiley, 2005 (cit. pp. 423, 425).

[120] Stefan Stanciulescu, Sandro Schulze, and Andrzej Wąsowski. “Forked and integrated variants in an open-source firmware project”. In: *ICSME*. 2015 (cit. pp. 396, 397).

[121] Mark Staples and Derrick Hill. “Experiences adopting software product line development without a product line architecture”. In: *APSEC*. 2004 (cit. p. 397).

[122] Pia Stoll, Len Bass, Elspeth Golden, and Bonnie E. John. “Supporting usability in product line architectures”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09. 2009 (cit. p. 410).

[123] Daniel Strueber, Mukelabai Mukelabai, Jacob Krueger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. “Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems”. In: *23rd International Systems and Software Product Line Conference (SPLC)*. 2019 (cit. p. 426).

[124] Mikael Svahnberg and Jan Bosch. “Evolution in software product lines: Two cases”. In: *Journal of Software Maintenance* 11.6 (Nov. 1999), pp. 391–422 (cit. p. 410).

[125] Yasuaki Takebe, Naohiko Fukaya, Masaki Chikahisa, Toshihide Hanawa, and Osamu Shirai. “Experiences with software product line engineering in product development oriented organization”. In: *SPLC*. 2009 (cit. p. 410).

[126] Christian Tischer, Andreas Muller, Thomas Mandl, and Ralph Krause. “Experiences from a large scale software product line merger in the automotive domain”. In: *SPLC*. 2011 (cit. p. 410).

[127] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007 (cit. pp. 407, 409, 410, 425).

[128] Martin Verlage and Thomas Kiesgen. “Five years of product line engineering in a small company”. In: *ICSE*. 2005 (cit. p. 410).

[129] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. “Towards a UML profile for software product lines”. In: *Software Product-Family Engineering*. 2004 (cit. p. 413).

[130] Tewfik Ziadi and Jean-Marc Jézéquel. “Software product line engineering with the UML: Deriving products”. In: *Software Product Lines*. Springer, 2006, pp. 557–588 (cit. p. 413).

[131] Roman Zippel. *KConfig*. Technical Documentation. 2017. URL: http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt (cit. p. 401).

Copy and paste is a design error.
David L. Parnas

12 Feature Modeling

An important notation for expressing domain models is *feature models*. Feature models are a simple, tree-based modeling notation that allows features and their constraints to be expressed. The latter restrict the valid combinations of features or express relations among features.

Feature models can nowadays be seen as the most successful notation to model the common and variable characteristics of products in a software product line. Proposed almost three decades ago, as part of the feature-oriented domain analysis (FODA) method [24], hundreds of variability management methods and tools have been introduced that build upon feature models. However, building feature models is still a creative process that requires expert domain knowledge. As such it is mainly done by humans, who need support in terms of a methodology—the focus of this chapter.

We already described the feature-modeling notation in Sect. 11.4. In this chapter, we will show how to systematically engineer feature models using a modeling methodology. But first, we will discuss the notion of feature and the different usages of feature models to set our modeling methodology into more context.

12.1 The Notion of Feature

Feature models center around the notion of feature [14]. Features are abstract entities used in a multitude of contexts, including software configuration, product marketing, scoping, requirements engineering, and domain analysis. As opposed to implementation assets (e.g., source files or components), features are more intuitive and domain-oriented entities understood by a range of stakeholders, not only developers. Features often also cross-cut software assets. For instance, the feature `ACPI` (Advanced Configuration and Power Interface), which controls power consumption in the Linux kernel, is a highly scattered feature, modifying many different parts (via `#ifdef` code fragments) of the source code in the kernel [37].

The notion of features is vague, which is in fact a core strength of features, since organizations can choose their own definition. In the most general sense, we can say that features abstractly represent functional or non-functional concerns of a software system. We can also see features as end-user-visible characteristics of a system [24, 14], or as distinguishable characteristics of a concept that is relevant to some stakeholder in the project. For example, choosing a manual or automatic transmission, when buying a car, might be interpreted as deciding a feature. Furthermore, features are a kind of concern. They are also a high-level requirement. Pragmatically, some organizations call the headlines of the requirements documents features.

In the literature, as many as 37 definitions of “feature” exist. Some definitions only capture the development side, e.g., when a feature is defined as a set of requirements [15], others only the business side [39]. As such, we recommend that you and your organization agree on the notion of feature. The following is a definition that, in our opinion, captures the notion of feature used in this book well, and can provide the basis for a more concrete definition that an organization can formulate.

Definition 12.1. *A* feature *is a concept in a domain. It can be seen as a high-level requirement. A feature represents commonality or variability in a product line. It is a unit of communication among stakeholders.*

12.2 Documenting a Feature

What we usually see of a feature is its name and its position in the hierarchy, which gives it some semantics. Mapping a feature to source code assets or to models gives it even more semantics. In practice, organizations need to track more information about features to engineer them. We call them feature facets.

Much information about these feature facets is distributed in different assets of a project and can be recovered. However, it makes sense to record such information. We coined the term feature facet for this reason [14]. You can either record various feature facets when defining the feature, or you can retroactively recover such facets [28].

Based on our previous work, and the book of Apel et al. [5], we suggest that recording the following information per feature is useful, but of course you need to tailor this list to your circumstances.

- **organizational information**, including organization level and responsible party (a.k.a. feature owner);
- **description**, including rationale, nature, and corresponding requirements;
- **relationship to other features**, including hierarchy, grouping, and constraints;
- **domain-specific dependencies**, including dependency on hardware, regulations, and runtime environments;
- **planning information**, including priority, costs, effort, and process;
- **realization information**, including scope, architectural responsibility, binding time, and behavioral specifications (e.g., invariants and conditions);
- **usage information**, including interested stakeholders, configuration knowledge (e.g., default activation) and questions, known effects on non-functional properties, and potential feature interactions;
- **quality assurance information**, including testing and approval process.

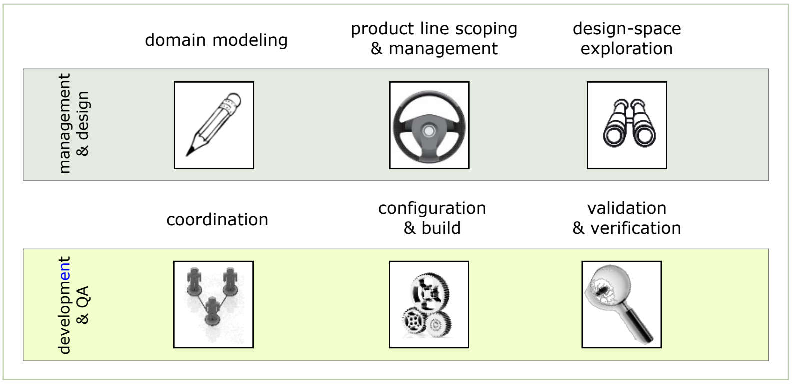


Figure 12.1: *Range of feature-model usages*

Exercise 12.1. Imagine the company UpAndDown that produces elevator systems. It provides customized solutions for private and public customers.

Analyze the domain. Which features are likely to be requested by many customers? Which features are likely to be requested by only a few customers? Which features could distinguish your products from the products of your competitors in this market segment?

Model the domain with a feature model. Pay attention to feature dependencies. Consider a maximum of around ten features.

12.3 Uses of Feature Modeling

As illustrated in Fig. 12.1, feature models are used for different purposes. We distinguish management & design uses, such as for domain modeling, scoping and managing the product line, as well as performing design-space exploration, from development & quality assurance (QA) purposes, such as coordination, configuration & build, and validation & verification.

A feature model can play the same role as a DSL model in a model-driven product architecture. A feature model can be used to derive a desirable product configuration, which can be fed into the code generator, to drive the *derivation* of an implementation of a specific product. In this sense, a feature model is an extremely simple meta-model, which describes its models—configurations adhering to the constraints of the feature tree.

Feature-modeling languages are used by several commercial and open-source product line tools such as pure::variants from the company pure::systems [38], Gears [26] from the company BigLever, or the open-source tool FeatureIDE [43]. Many configuration languages grown internally within various projects resemble feature modeling a lot. Recall our discussion of the Linux kernel's language Kconfig in this chapter.

12.4 A Feature-Modeling Process

Now that we discussed the feature-modeling notation in detail, and also took a look at the feature-modeling-like language Kconfig used in the Linux kernel and other systems software projects [13], let us look at the modeling process itself. In this section, we will refer to our modeling principles

Symbol	**Description**
ⓘ?	Decision affecting following activities
⚙	Activity
(⚙)	Optional activity
⚙⚙	Composite activity
(⚙⚙)	Optional composite activity
⚙	Sub-Activity of a composite activity

Table 12.1: Legend for feature-modeling guidelines

presented in Nesic et al. [36] and present core questions you need to answer as well as different kinds of actions you need to perform. Tbl. 12.1 explains core terms and their icons we will use in this section.

The following process and principles should be applied when creating a feature model for a software product line. If you are using feature models just for brainstorming or other creative phases of software engineering without the goal of creating a product line at some point, you probably do not need to consider the following modeling process and principles.

There are three ways of adopting a product line [25, 12], and they influence the way you create the feature model. When building a product line from scratch, also called *pro-active adoption*, you predominantly create the feature model in a top-down fashion. From domain analysis and scoping, where you model the domain in a reasonable scope—for instance, you model the features that you think you can develop and sell to customers—you start by creating the top-level features and then refine them. When building a product line from one existing product, also called *re-active adoption*, or from multiple existing products, also called *extractive adoption*, you predominantly build the feature model in a bottom-up fashion. From the existing products you have configuration options, which you model as optional features as leaves. From the differences between existing products you identify differences and try to understand why these differences exist from a domain perspective, and these differences you model as features. However, while we say "predominantly" top-down or bottom-up, in all three adoption scenarios one does both (principle M_5: Use a combination of bottom-up and top-down modeling), as we discuss in the phase Domain Analysis and Scoping for the activity Feature Identification on page 444.

In our process, we classify the different activities into four phases: Pre-Modeling, Domain Analysis and Scoping, Modeling, and Maintenance and Evolution. Figure 12.2 depicts these phases, together with typical iterations among the last three phases.

Pre-Modeling Activities

In the first phase, before you start the actual modeling, you plan the feature modeling and train the relevant stakeholders. The result is a description of the model purpose, a clarification of the stakeholders involved and their roles, and a change and expectation management plan. We recommend

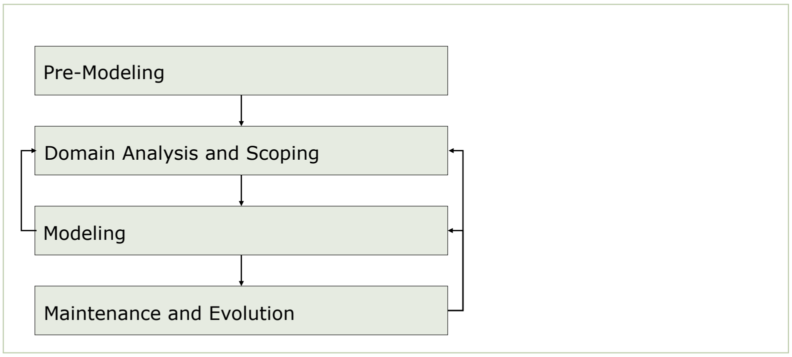


Figure 12.2: *The four main phases of the feature-modeling process*

defining the model purpose (activity Define Model Purpose) and training (activity Provide Training) in iteration, which allows the purpose to be clarified and refined.

Define model purpose. Your first activity is to clarify what to use the model for (principle PP_3: Define the purpose of the feature model). You need to do that in order to focus the modeling on the relevant features and modeling concepts (e.g., constraints), and avoid wasting time on irrelevant ones. Choose among the different uses shown in Fig. 12.1. However, note that when the feature model should serve both management & design and development & QA purposes, there is often a tension between designing the model more towards capturing domain- and business-oriented features or towards implementation-oriented features. In other words, the feature model is often seen as a pivotal model artifact, used as a communication platform to support business goals, while at the same time it should be possible to map the features to describe the software assets and control the platform, that is, be able to derive individual products in an automated process supported by a configurator tool, for example.

Identify stakeholders. Your second activity is to identify the relevant stakeholders (principle PP_1: Identify relevant stakeholders), who can have diverse roles in your organization. We distinguish between three kinds, which are not necessarily disjoint:

- (i) *experts* are those who will provide input about features and their constraints, as domain- or implementation-oriented experts;
- (ii) *modelers* are those who will perform the modeling; and
- (ii) *model users* are those who will use and benefit from the feature model.

The *experts* (i) should have sufficient knowledge about the domain (i.e., know the problem space) or about the implementation (i.e., know the solution space). While the former understand what features need to be developed for economic benefit (e.g., business and sales experts), the latter know the technical details about the software in depth (e.g., developers). Depending on the purpose of the feature model, you want to have a representative of one of each kind, multiple representatives of either kind, or one who has knowledge about the domain and the implementation. In our experience, we even observed companies where the developers traditionally had very good insights into the business and sales aspects, especially when there used to be a close relationship due to frequent meetings. In many cases, however, developers have never learned to think in terms of the domain and business and require training and a pilot project to obtain such a perspective.

The *modelers* (ii) are often system and software architects, project managers, or requirements engineers, since they usually build abstract system models. Our experience shows that the number of stakeholders performing the modeling in an organization should be low, perhaps as low as a single person (principle PP_6: Keep the number of modelers low).

Finally, it is important to decide who are the *model users* (iii). If they are end-users or even customers, then the feature model needs to be understandable. It is also necessary to model all the constraints among features (principle D_2: If the main users of a feature model are end-users, perform feature-dependency modeling). This way, the configurator tool can ensure that only correct configurations are created and valid variants are derived from the platform. When a domain expert configures the model, who knows all the details about features and constraints among them, then it might not pay off to invest the effort of modeling constraints (see below where we talk about dependency modeling).

Provide training. Training should include becoming familiar with the feature-modeling notation and the tool used, as well as with the process and principles of feature modeling—a sub-activity we call **Tool and Notation Training**. Training involves familiarization with product line engineering (e.g., platform architectures, software configuration, and product derivation), perhaps with this book chapter—a sub-activity we call **SPLE Education**. To make it easy for learners to understand the feature-modeling notation and its semantics, ideally it can be related to concepts they are used to intuitively. For developers, feature types and their graphical representation can be related to classes or data types. For instance, a feature with a checkbox is a feature of type Boolean. In practice, the training is often done together with a tool vendor.

We recommend that training involves a **Pilot Project** of around three days (principle T_3: Conduct a pilot project). This should be done with a small (sub-)system of the company that exists in multiple variants, which have sufficient commonality and do not come with strict deadlines regarding the release to production. This allows very fast feedback loops and facilitates training. If your organization does not have an existing system and rather wants to adopt SPLE from scratch, then you can refer to existing data sets of clone & own-based systems [27, 42, 29].

The pilot should comprise all the activities of the feature-modeling phases, which we explain shortly: the modeling phase as well as the maintenance and evolution phase. We recommend to create a platform with around 20–50 variation points that represent the differences in the individual variants. So, identify the differences in the implementations, abstract them into the respective features, and model them in a feature model, as we explain for the feature-modeling phases shortly.

As a guided exercise, a core benefit of performing a pilot project is to "walk" those who have detailed knowledge about the variant implementations up to the domain. Those stakeholders usually understand the differences in detail, that is, in terms of implementation concepts. When asked about the details, they usually provide those implementation-level details. The idea is to ask them various times (cf. principle M_3) why the differences exists, leading to increasingly domain-oriented explanations, until the difference can be described by the presence or absence of a specific feature, as a domain-oriented concern. The pilot project will also give experience in product derivation (cf. principle QA_2). Engineers can experience whether the derivation feels viable, that is, go through the feature model and make selections to establish a configuration.

The pilot project also helps to, if envisioned, connect the business and development worlds. Connecting features to assets and to business aspects is also important, since doing that later is difficult. This will also improve acceptance of the feature model, since product derivation before was usually a manual and error-prone activity, requiring copying and pasting software assets and packaging them properly. Selecting a reasonably small subsystem for the pilot project can substantially improve feature-model training and acceptance.

Create change and expectation management. When an organization wants to introduce feature modeling and SPLE, defining and executing a communication plan is crucial. The plan should explain the benefits, especially the reuse potential and the respective business-related benefits, such as shorter time to market. We recommend describing the benefits tailored to the different stakeholders. For instance, the stakeholders who are more business-oriented benefit from having features, from having them organized in the feature model, and from having feature descriptions. The more development-oriented stakeholders benefit from clear feature requirements, which the features are mapped to, as well as keeping an overview understanding of the development.

The communication plan should also explain the necessary changes in the process and in the organizational structure, as well as in the architecture of the platform and the individual products. Explaining the notion of feature, and why we need features, is also important.

Establish a forum and a workshop format. It is also advisable to establish a forum with regular meetings to discuss maintenance and evolution. Since a feature model is brittle, one or a few stakeholders in the organization should become the main modeler(s), to be consulted in those forums.

To elicit information about new features and their relations, a workshop format should be adopted. The workshops help to elicit information from the stakeholders (principle IS_1: Rely on domain knowledge and existing artifacts to construct the feature model), to validate the model (principle QA_1: Validate the obtained feature model in workshops with domain experts), as well as to evolve and maintain it.

It is also advisable to put an approval process for new features in place, ideally as part of the workshop format.

(⚙) *Define decomposition criteria.* This optional activity aims at defining some criteria that help modelers decide how to decompose features in the model (principle PP_4: Define criteria for feature to sub-feature decomposition). As discussed in the box "The Feature Hierarchy" on page 449, the meaning of the hierarchy edges in a model is intentionally not well defined. Modelers are relatively free to stick with Part-Of or Is-A relationships between features and model the hierarchy freely to be as intuitive as possible, or to conceive and document domain-specific decomposition criteria for the model. These could reflect existing hierarchies (e.g., of physical parts of the product) in the organization or even parts of the architecture decomposition, or other hierarchies that your stakeholders are familiar with in customer-facing catalogs.

(⚙) *Unify domain terminology.* This optional activity can be necessary when the domain terminology is too diverse and ambiguous in the organization (principle PP_2: In immature or heterogeneous domains, unify the domain terminology). The risk is that different perceptions of domain concepts might cause confusion among stakeholders and lengthy discussions. We suggest you provide a dictionary with descriptive terms for feature names. If several feature models will be created, you could also define a hierarchical naming schema and prefixes for features in particular (sub-)models. A common language is the precondition for successful joint work among the stakeholders involved.

Domain Analysis and Scoping Activities

After the pre-modeling phase, there are two main phases carried out iteratively (principle PP_5: Plan feature modeling as an iterative process). In the first one, described in this subsection, you extract information about features and their relationships relevant to the subsequent modeling phase. Iterating between the two phases allows you to gradually increase your modeling expertise, as well as to safely and incrementally evolve the feature model.

The idea is that you start with an initial domain analysis and scoping, to gather and document information (mainly a list of features and their relationships) in a way that is sufficient to proceed with the modeling activities. Then you iterate—increasingly more closely—where you obtain features and immediately model them. You usually even develop the system in parallel. Once you have an initial software system controlled by the feature model, this will also help with the iteration.

We recommend to perform the activities of this phase in workshops (principle M_1: Use workshops to extract domain knowledge). A workshop is usually the best way to start to obtain core domain knowledge from the relevant experts. Recall the activity Establish a Forum and a Workshop Format on p. 443 above.

⚙ *Identify features.* Before modeling features, we first need to identify them. We distinguish between the bottom-up and the top-down strategy. The former you mainly apply for the extractive and the re-active adoption of product lines, so when you already have a system or a set of cloned system variants. The latter you apply for pro-active adoption, when you need to decide what features to realize and how to organize them. In practice, you apply both the bottom-up and the top-down strategies, but put more emphasis on either one based on the adoption strategy. You should also recall how a feature is defined (cf. Def. 12.1), and what its main characteristics are—most importantly, that a feature represents a distinct, well-understood, and graspable aspect of the software system (principle M_6: A feature typically represents a distinctive, functional abstraction).

When identifying features, you should first focus on those that distinguish variants (principle M_2). You should also prefer features of type Boolean (principle M_{10}) for easy comprehension of the resulting feature model. The following two main identification strategies exist:

- ⚙ **Bottom-Up Feature Identification** If you have one existing system (re-active adoption), you start by considering the existing and demanded configuration options, which give you a list of features to start with.

  When you have existing system variants (extractive adoption), which often arise from clone & own, then you perform pairwise diffing. You can use a standard diffing tool, such as the one that is built into Eclipse, Notepad++ with the Compare plugin, or the tool Meld,[1] which provides more extensive diffing support. Specifically, perform a pairwise diff among the variants, which means that you take one as a base and diff it with another one. You observe the differences, then try to understand why these differences are there, in order to identify features. Of course, to come up with a product line, you need to convert the differences into variation points using a suitable variability mechanism. We refer to Sect. 11.3 and the relevant literature (Apel et al. [5], Chapters 4 and 5) for details about implementation techniques for variation points, as well as for methods and tools to integrate the cloned systems into one platform [6, 30, 40, 27].

  Overall, your task is to "convert" implementation differences into features. The idea, which we already explained for the pilot project above, is to understand why a difference exists. A typical technique (principle M_3: Apply bottom-up modeling to identify differences between artifacts) is to ask those with detailed variant implementation knowledge various times why the difference exists; this leads to increasingly domain-oriented explanations, until the difference can be described by the presence or

[1] https://meldmerge.org

absence of a specific feature. In other words, you lift the implementation-level differences to the domain.

- **Top-Down Feature Identification** This sub-activity is usually the responsibility of dedicated domain analysis [23] and product-line-scoping methods [41, 21, 22]. According to Czarnecki and Eisenecker [16], the *"purpose of Domain Analysis is to select and define the domain of focus and collect relevant domain information and integrate it into a coherent domain model."* The domain model in our case is a feature model. Product-line-scoping methods, such as PuLSE-Eco [7], systematically select and prioritize the features that an organization wants to realize. These should bring an economic benefit for the organization and be in line with its business strategy (e.g., considering vision, strategy, finance, and commercial aspects).

After identification, the feature needs to be approved in some way by your organization. This approval process can be part of the established forum and workshop format (cf. page 443). Once approved, you can add it to the feature model (see activity Add Features below). It should also be documented (principle M_{11}: Document the features and the obtained feature model).

The next question you should think about is whether you need to identify and model cross-tree constraints between features. Many constraints will already be reflected in the feature hierarchy and in feature groups, or as mandatory features. In any case, these constraints need to reflect the semantics of how you can combine features via the assets they map to. For instance, when you combine features into an OR group, but the system does not build or crashes when you select more than one of these features, then an XOR group would properly constrain the features. Beyond these constraints, which are easily visible in a feature model, you need to decide whether you need to model cross-tree constraints, which are often more intricate and challenge comprehension of the feature model.

Two modeling principles come in handy for making this decision. If the model is configured by (company) experts, avoid modeling of cross-tree constraints (principle D_1). Since it is very expensive to accurately model all constraints, and since the experts will likely know all the constraints, it usually will not pay off to model them. Some case studies [10] shed more light on this issue. First, you often need a consultant to help the customer to decide which features are needed, so you can often save the effort of modeling constraints. Another strategy seen in practice is to maintain sets of tested configurations, which are evolved and maintained together with the model. In contrast, if the main users of the feature model are end-users, then you need to model the cross-tree constraints (principle D_2). This can easily be seen in the Linux kernel (cf. Sect. 11.2) and many other systems software projects [13]. The complexity of these models and the sheer number of their configurations used for running systems demand that all constraints should be modeled.

(⚙) *Identify constraints.* As discussed in the box on page 450, constraints restrict the possible configurations of a feature model, to prevent undesired or invalid system variants, and to enhance the configuration experience.

But where do the constraints come from? All systems composed of parts (in our case, software assets) have constraints over those parts, arising from domain, marketing, or technical restrictions. Since we abstract the selection of those parts to the selection of features (i.e., we mapped the parts to features), what we do is to lift those constraints over parts to constraints over features, which is not always trivial.

- **Code Constraints** Empirical studies show that in systems software, up to half of the constraints in a feature model can be found in the codebase and extracted using various program analysis techniques [34, 35]. Since such analysis techniques are difficult to set up and use, the developers should rather inform the modelers about such constraints or declare them directly in the model.
  We distinguish between two major kinds of sources: the so-called *feature effect* and the prevention of *build- or runtime errors*. While details are described by Nadi et al. [35], intuitively, feature effect refers to the idea that enabling a feature in the model should have an effect on the resulting variants. In other words, if you enable the feature and nothing will happen, then likely some constraints are missing. A typical example is a feature whose implementation (i.e., the variation point controlling inclusion of the implementation) is contained in that of another feature. Of course, if the latter is disabled, enabling the former will not have any effect. So, *feature effect* means that enabling a feature should lead to a lexically different program or to one that behaves differently. The other source of constraints aims at the prevention of *build- or run-time errors*, and is also described in more details by Nadi et al. [35]. Such errors can occur early when the system fails to build, that is, fails to preprocess, parse, compile, type-check, or link. They can also occur late at runtime, for instance, when the system crashes due to null-pointer de-referencing or buffer overflows. Notably, they are much more difficult to detect than build-time errors.
- **Domain Constraints** Such constraints arise from domain knowledge and are usually not contained in the codebase. Examples are dependencies among hardware devices, which are instead contained in documentation or in the experience and knowledge of domain experts or developers. To some extent, these constraints can be found through testing the different combinations of hardware and then adding them. However, mostly they need to be provided by the domain experts.
- **Other Constraints** Further sources are marketing experts, who might want to limit feature combinations for business reasons, or to simplify feature selection for the customer. Constraints can also be used to partially configure a feature model, which is called staged configuration [18]. Finally, some feature-modeling tools allow specification of soft constraints, such as “recommends” [10].

From these sources of constraints, observe that, while code constraints are reflected in the codebase and could in principle be recovered, the other sources illustrate that feature models contain unique knowledge.

Finally, when identifying constraints, it is normal that initially you are not aware of all the dependencies. In fact, it is often difficult to see them early on, which can also be seen in the Linux kernel [31]. There, when developers add new features, it is sometimes observable that they fix the dependencies in several subsequent commits.

Finally, after identifying the constraints, document them together with their rationales (principle M_{11}: Document the features and the obtained feature model).

Modeling Activities

In the modeling phase, the goal is to obtain a feature model based on the documented information about features and relationships in the previous phase (Domain Analysis and Scoping Activities).

? A core question to begin the modeling with is *whether you want to physically separate the partitions of the envisioned model into different feature-model files or not* (principle MO_3: Split large models). If so, perform the following two activities, otherwise continue with activity Define Coarse Feature Hierarchy below. Still, even if you do not want to decompose and rather want to create one feature model, it can be beneficial to temporarily decompose into models representing different stakeholder-related features, to model them in isolation, and later integrate them.

() *Model modularization.* Decomposing a feature model into smaller ones has pros and cons. It facilitates distributed, independent evolution and maintenance of the model, eases version management, as well as discourages (or limits) cross-tree constraints across the models. However, it also raises consistency maintenance issues. In contrast, not decomposing avoids the overhead of having to maintain multiple model files and their inclusion in a central one, but large models quickly become unmanageable.

Whether you should decompose depends on multiple factors. First, it depends on whether you find an easy decomposition of the feature-model hierarchy into coherent sub-trees. For instance, a sub-tree could contain features representing implementation details and another one those representing user-visible characteristics. Other factors are the estimated software size and estimated number of features. From our experience, large models with several hundreds of features are all modularized into multiple files. The Linux kernel with the distribution of its ultra-large model across 1,000 files (cf. Sect. 11.2) is an extreme example. From our experience, all commercial models we have seen with several hundred features were all split into multiple ones. The hierarchy of feature models sets up the first framework for the platform—it is an initial structure that helps with the modeling. This hierarchy can be distributed along the codebase (i.e., as in the Linux kernel) or organized in a dedicated folder structure.

The Feature Hierarchy

The feature hierarchy is one of the most valuable parts of a feature model. It organizes knowledge, thereby helping stakeholders to keep an overview understanding of complex systems in terms of features.

The meaning of the hierarchy edges in feature models is not explicitly defined from a domain perspective. In our experience, they most often resemble a *Part-Of* relationship, but can also be of the *Is-A* kind (a.k.a. generalization), so rather expressing ontological relationships. *Part-Of* also makes sense from a configurator perspective. Recall that a child feature implies its parent in the semantics. You want to avoid selecting a feature without it having an effect, to avoid meaningless configurations (redundant feature selections that do not change the actual derived variant). When an asset that is controlled by a child feature is part of another asset controlled by the parent feature, then you should not be able to enable the contained asset, which will never be there, since the container is missing.

A good feature hierarchy has the following properties:

- It is intuitive and easy to navigate.
- It abstracts over the codebase (folder) hierarchy.
- Its top-level features are more abstract and business-oriented. Those in the middle levels represent functional aspects. The bottom-level features are usually more detailed technical concerns (e.g., hardware, libraries, diagnostics, and configuration options.
- It organizes features into sub-trees that logically partition the domain.
- Its organization reduces cross-tree constraints, thereby increasing cohesion and reducing coupling.
- It does not have a deep hierarchy. In practice, hierarchies have 3–6 levels. The maximum depth we have seen (in the Linux kernel) is 8. Deep hierarchies have many intermediate features, which are usually vague and not very distinct, and as such difficult to understand for stakeholders.

Model modularization has two sub-activities:

- **Define Structure of Model Files** To decompose, you define a hierarchy of feature models, beginning with a root model. This model's top-level features then become root features in the decomposed model files. You carry out this sub-activity at the beginning.
- **Maintain Consistency Between Model Files** To maintain consistency, you find features that participate in dependencies across the models, and then move them into a separate "interface" feature model. This practice isolates the inter-model dependencies and eases their maintenance. You carry out this sub-activity during the actual modeling once you feel that the cross-model dependencies are getting out of hand.

Define coarse feature hierarchy. You start by creating an initial, coarse hierarchy of features within the feature model (if you created multiple feature models, select the one whose features you think are most well-understood).

Start by defining feature groups, where you model features that belong to a horizontal domain or have a close relationship. Think how to navigate those groups and existing features in a better way. You maximize cohesion

Feature Constraints

Constraints restrict the values of features based on other features' values to prevent undesired or invalid variants. Or, in other words, constraints restrict the possible configurations (and, thereby, system variants) of a feature model. Most of these constraints should be reflected in the feature hierarchy and in feature groups, or by making features mandatory. The remaining constraints are added as cross-tree constraints.

Constraints exist for various reasons [35]:

- Constraints enforce low-level dependencies between software assets, mainly code. Since software systems, especially product lines, are built modularly and have variation points, features might need to use other features to function. For instance, there can be a definition-use relationship, such as a method definition provided by the assets of one feature, and called from within the assets of another feature.
- Constraints assure a correct runtime behavior—mainly since some dependencies for features might only be known or available at runtime. For instance, in the Linux kernel, many driver features rely on the availability of certain hardware or interfaces (e.g., communication ports) only available for a certain hardware architecture. So, there would be a dependency to, for instance, the feature X86.
- Constraints improve the user's configuration experience. As an input to interactive configurator tools, feature models facilitate configuration, when shown as menus and sub-menus in a tree-like organization. To foster such an organization, feature models contain constraints. Interestingly, when configurator tools do not offer intelligent choice-propagation or conflict-resolution support, such as the Linux kernel configurator, often additional constraints are needed to compensate for the lack of such a support.
- Constraints avoid corner cases of feature combinations. Given the sheer number of possible configurations and ways of combining features, often undesired feature interactions [4] arise, which need extra code to handle them. For instance, we observed that in the Linux kernel, when supporting a certain, rare combination of hardware would be too expensive, developers might decide to disallow such a corner case via constraints. Some systems even provide a disabled feature *Broken* that features not currently supported can depend on.

and minimize coupling with feature groups (principle MO_5). Specifically, feature groups should represent related functionalities—these are within a group, while there is low coupling to other groups (so, no cross-tree constraints). In contrast, you use abstract or mandatory features (cf. Def. 11.8) for structuring the overall model.

Another idea is that you organize features into sub-trees that logically partition the domain. Thereby, you try to reduce the need for cross-tree constraints across those partitions (sub-trees), but rather keep constraints within them. In other words, you try to increase cohesion and reduce coupling.

To form the hierarchy, consider the properties given in the box on p. 449. It is probably useful to recall that the top-level features are more abstract and business-oriented (principle MO_2: Features at higher levels in the hierarchy should be more abstract), so that they can be communicated to customers. Intermediate features (i.e., those in the middle levels) represent functional aspects. Towards the leaves, the features are more technical—often, you create a domain- and business-oriented feature and then, when actually implementing it, need to add more specific and perhaps technical sub-features. You try to avoid having many intermediate features, which are usually vague and difficult to understand for your stakeholders.

After defining a coarse hierarchy, it will be iteratively refined in the next activity (Add Features).

⚙ *Add features.* While identifying features, you extend and refine your feature model. The new features you identify will either already exist in the feature model, or you need to add the newly identified features at relevant places in the feature model.

Since you always want to limit the number of features, you should first look for features that are similar and ask yourself whether an existing feature can be adjusted. You also do that because there is always the cost of a new feature to consider, and you want to avoid a growing pool of features. So, you first try to update and enhance existing features.

When placing the feature in the hierarchy, consider again the properties given in the box on page 449. However, the location should still "feel right" to the involved stakeholders, and as such, a discussion among them might be necessary.

Finally, define the relevant meta-data (e.g., feature title and short description); especially define default feature values (principle M_8), which substantially eases creating a feature-model configuration (making deriving a configuration a reconfiguration problem). Further meta-data that might be relevant in your organization could be the rationale why the feature was added, the feature owner (if this role exists) or party responsible for the feature, or so-called visibility conditions [13], determining when the feature is even visible to the user when creating a configuration.

(⚙) *Model constraints.* If you decided to identify and model constraints (recall the question on p. 446), then conduct this optional activity.

Declaring dependencies between features might require regrouping of features, removing the dependency, or extracting the dependencies into an interface feature model (principle MO_3: Split large models). So, you should always evaluate whether you really need to define those dependencies.

You should avoid complex constraints, which typically come in the form of Boolean expressions. Such constraints challenge comprehension, maintenance, and evolution of the model (principle MO_4: Avoid complex cross-tree constraints). You first try to model constraints using the feature hierarchy and other graphical elements from feature models (e.g., mandatory features or feature groups). In fact, an indicator of a good feature hierarchy is a low ratio of cross-tree constraints. If you still cannot restrict the remaining cross-tree constraints to simple binary dependencies (e.g., *required* in the form of an implication between two features, or *excludes* in the form of an implication by a feature of the negation of another one), you can also put some constraints into the presence conditions of the variation points, which keeps the model clean at the cost of a slightly more complex mapping between features and software assets (i.e., variation points).

The source of the constraint (cf. activity Identify Constraints) gives you an indication how to model it. Interestingly, constraints arising from the source we called *feature effect* are mostly reflected in the feature hierarchy. This makes a lot of sense when you remember that a feature always implies its parent in a feature model, enforcing that the sub-feature has an effect. Constraints preventing build- and runtime errors are rather seen in cross-tree constraints or feature groups.

( ) Define views. In addition to model modularization, some feature-modeling tools allow creation of views, for instance through filters or partial configuration, sometimes also called profiles (principle M_9: Define feature-model views).

Validation. After the modeling activities, it is time to check that the modeling was correct in the eyes of the stakeholders. After changes during evolution and maintenance, you also want to use the following ways of validation, especially the last one, regression testing.

- **Stakeholder Reviewing** In the workshop format established during the planning phase, various stakeholders should be invited to validate the feature model (principle QA_1: Validate the obtained feature model in workshops with domain experts). We advise that different domain experts participate, given their individual area of expertise. They can validate that the right features and constraints were identified and modeled correctly, and they can advise on feature names and whether the structure of the hierarchy is intuitive. It is also beneficial when experts who did not participate in the modeling take part in the validation—among other things to comment on the intuitiveness of the feature model.
- **Perform Product Derivations** When one of the purposes of the feature model is to support product derivation, you should let the relevant stakeholders perform it for some example variants (principle QA_2: Use the obtained feature model to derive configurations). This can be done in the workshop format established. Obviously, the experience will be different than before, which was mostly manual. So, the stakeholders will select features in a certain order, and by doing so, they will be able to tell the modeler whether it feels right and whether it will be effective. As for which variants to derive, you should do that for existing ones, but also derive at least one that never existed before, which reinforces the benefit of having a platform with automated variant derivation through feature-model configuration.
- **Regression Testing** When iteratively creating the feature model, as well as maintaining and evolving it, you can easily break existing configurations. Many of the established feature-modeling tools, including FeatureIDE [32], will provide you some automated analysis that tells you whether a change to the model will have an effect on existing configurations. These analyses are confined to the feature model, but it is often desired to analyze the effect on the actual variants [33]. This

requires creating regression tests (principle QA_3: Use regression tests to ensure that changes to the feature model preserve previous configurations) using typical testing methods (e.g., unit tests), but these should be given different configurations, ensuring the coverage of feature configurations that cover variants that are in use, ideally on the customer side. Knowing those requires either tracking such configurations or obtaining expert knowledge from the developers implementing the respective software assets. For instance, a developer usually knows from experience which features might interact and should be tested for certain modules.

Maintenance and Evolution Activities

To evolve the model, you can apply the activities from the previous two phases: Domain Analysis and Scoping Activities, as well as Modeling Activities. Especially the established workshop and forum (recall the respective planning activity on page 443) come in handy here. Still, while many stakeholders are involved, one or only a few of them should ultimately control the model and make changes (principle MME_1: Use centralized feature model governance). Feature models are brittle assets and need to be evolved with care, to avoid inconsistencies that would have an impact on many different variants. In this light, it is also important to regularly perform the validation activities (cf. page 452).

The following activities additionally support evolving the model, as well as maintaining it.

⚙ *Model version control.* Tracking the evolution of the feature model, with the ability to go back and analyze it, is core. There have been attempts at supporting the versioning at the feature level, but according to our experience, you should version the feature model in its entirety (principle MME_2). While keeping an overview with a more fine-grained way of versioning is already difficult, the main reason is probably that individual features are not units of deployment or packaging, but whole system variants are. As such, it is more relevant to go back to such whole snapshots instead of individual feature versions.

⚙ *Remove features.* Performing this activity is necessary from time to time, but surprisingly difficult. Many companies therefore avoid removing features. However, for very long-living platforms, removal is absolutely necessary, to reduce the maintenance overhead and system complexity.

The removal of features should be discussed in the established workshop or forum format. Once decided, a strategy is to remove the feature step-wise. If supported by the feature-modeling tool, the feature should first be flagged as deprecated, and also its default value should be changed to false. The next step is to make the feature a dead feature via constraints, so that it cannot be selected anymore. The final step is to remove the feature from the model, and also the respective software assets.

Some companies even model the overall lifecycle states of a feature internally as a state machine, with around 5–8 states. A good example of states is: Proposed, Approved, Implemented, Deployed, Obsolete, Decommissioned. The state Obsolete would comprise the above steps of removing the feature from the model, while in the state Decommissioned, the feature is removed from the model and assets.

⚙ *Optimizations.* Of course, over time, the constraints become more intricate, and the hierarchy might not be as intuitive as necessary. So, an important activity is to optimize the hierarchy and the constraints. However, without proper tool support for refactoring, it is relatively easy to invalidate existing variants, which should be avoided. Performing the validation activities is crucial (cf. page 452).

Further Reading

The body of work on feature modeling is humongous. The FODA report [24] is the most popular work on feature-oriented domain analysis, and has proposed the feature-modeling notation. Another introduction to feature modeling is chapter 4 of the book by Czarnecki and Eisenecker [16]. Nowadays, many different variants of the original feature-modeling notation exist. A brief history of these notations is provided by Berger and Collet [9].

It is worthwhile to look at feature models as they are used in practice. Our study of feature modeling in systems software, including the Linux kernel, sheds light on those models, especially on the languages that are used and the characteristics of models [13]. Our survey of variability modeling in Berger et al. [12] discusses how product lines are adopted and what languages, tools, and scales of feature models are used in practice. It is complemented by a qualitative study of cases in industry [10]. Other interesting and detailed descriptions of feature modeling comprise the report of Hubaux, Heymans, and Benavides [20] in the context of a re-engineering project, and the study of (feature-model-like) sales configurators by Abbasi et al. [1].

When many feature models exist, some explicit management might be needed. The works by Acher et al. [2, 3] describe techniques to de-(compose) feature models. There are also plenty of analyses on feature models, surveyed by Benavides, Segura, and Ruiz-Cortés [8] and revisited in 2019 [19]. Here, see a related investigation on analyses that are actually needed by industry, as a contrast in Mukelabai et al. [33].

Of course, various other variability-modeling techniques exist. An interesting comparison of feature models with so-called decision models by Czarnecki et al. [17] shows that there are actually many commonalities and only minor differences (e.g., the ability to model the commonality is given in feature models, but not in decision models). Furthermore, if you have not only product lines, but whole ecosystems, such as Android (cf. Sect. 11.1), you will need different management and especially modeling techniques for variability, such as manifest files. See our report in Berger et al. [11].

References

[1] Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. “The anatomy of a sales configurator: An empirical study of 111 cases”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2013, pp. 162–177 (cit. p. 454).

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. “Composing feature models”. In: *International Conference on Software Language Engineering*. Springer. 2009, pp. 62–81 (cit. p. 454).

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. “Familiar: A domain-specific language for large scale management of feature models”. In: *Science of Computer Programming* 78.6 (2013), pp. 657–681 (cit. p. 454).

[4] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. “Feature interactions: The next generation (Dagstuhl Seminar 14281)”. In: *Dagstuhl Reports* 4.7 (2014), pp. 1–24 (cit. p. 450).

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013 (cit. pp. 438, 445).

[6] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. “Reengineering legacy applications into software product lines: A systematic mapping”. In: *Empirical Software Engineering* 22.6 (2017), pp. 2972–3016 (cit. p. 445).

[7] Joachim Bayer et al. “PuLSE: A methodology to develop software product lines”. In: *Proceedings of the 1999 Symposium on Software Reusability*. 1999 (cit. p. 446).

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. “Automated analysis of feature models 20 years later: A literature review”. In: *Information Systems* 35.6 (2010), pp. 615–636 (cit. p. 454).

[9] Thorsten Berger and Philippe Collet. “Usage scenarios for a common feature modeling language”. In: *First International Workshop on Languages for Modelling Variability (MODEVAR)*. 2019 (cit. p. 454).

[10] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. “Three cases of feature-based variability modeling in industry”. In: *MODELS*. 2014 (cit. pp. 446, 447, 454).

[11] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. “Variability mechanisms in software ecosystems”. In: *Information and Software Technology* 56.11 (2014), pp. 1520–1535 (cit. p. 454).

[12] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. “A survey of variability modeling in industrial practice”. In: *VaMoS*. 2013 (cit. pp. 440, 454).

[13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “A study of variability models and languages in the systems software domain”. In: *IEEE Transactions on Software Engineering* 39.12 (2013), pp. 1611–1640 (cit. pp. 439, 446, 451, 454).

[14] Thorsten Berger et al. “What is a feature? A qualitative study of features in industrial software product lines”. In: *SPLC*. 2015 (cit. pp. 437, 438).

[15] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Pearson Education, 2000 (cit. p. 438).

[16] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000 (cit. pp. 446, 454).

[17] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. “Cool features and tough decisions: A comparison of variability modeling approaches”. In: *VaMoS*. 2012 (cit. p. 454).

[18] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. “Staged configuration through specialization and multilevel configuration of feature models”. In: *Software Process: Improvement and Practice* 10.2 (2005), pp. 143–169 (cit. p. 447).

[19] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. “Automated analysis of feature models: Quo vadis?” In: *Computing* 101.5 (2019), pp. 387–433 (cit. p. 454).

[20] Arnaud Hubaux, Patrick Heymans, and David Benavides. “Variability modeling challenges from the trenches of an open source product line re-engineering project”. In: *2008 12th International Software Product Line Conference*. IEEE. 2008, pp. 55–64 (cit. p. 454).

[21] Isabel John and Michael Eisenbarth. “A decade of scoping: A survey”. In: *Proceedings of the 13th International Software Product Line Conference*. 2009, pp. 31–40 (cit. p. 446).

[22] Isabel John, Jens Knodel, Theresa Lehner, and Dirk Muthig. “A practical guide to product line scoping”. In: *10th International Software Product Line Conference (SPLC’06)*. IEEE. 2006, pp. 3–12 (cit. p. 446).

[23] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 1990 (cit. p. 446).

[24] Kyo Chul Kang. “FODA: Twenty years of perspective on feature models”. In: *SPLC*. Keynote Address. 2009 (cit. pp. 437, 454).

[25] Charles Krueger. “Variation management for software production lines”. In: *Proceedings of the Second International Conference on Software Product Lines*. SPLC 2. 2002 (cit. p. 440).

[26] Charles W. Krueger. “BigLever Software Gears and the 3-tiered SPL methodology”. In: *OOPSLA*. 2007 (cit. p. 439).

[27] Jacob Krueger and Thorsten Berger. “Activities and costs of re-engineering cloned variants into an integrated platform”. In: *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. 2020 (cit. pp. 442, 445).

[28] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. “Where is my feature and what is it about? A case study on recovering feature facets”. In: *Journal of Systems and Software* 152 (2019), pp. 239–253 (cit. p. 438).

[29] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. “Getting rid of clone-and-own: Moving to a software product line for temperature monitoring”. In: *SPLC*. 2018 (cit. p. 442).

[30] Max Lillack, Stefan Stanciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. “Intention-based integration of software variants”. In: *41st International Conference on Software Engineering*. ICSE. 2019 (cit. p. 445).

[31] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. “Evolution of the Linux kernel variability model”. In: *SPLC*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 136–150 (cit. p. 448).

[32] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017 (cit. p. 452).

[33] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. “Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems”. In: *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018 (cit. pp. 452, 454).

[34] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Mining configuration constraints: Static analyses and empirical results”. In: *ICSE*. 2014 (cit. p. 447).

[35] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Where do configuration constraints stem from? An extraction approach and an empirical study”. In: *IEEE Transactions on Software Engineering* 41.8 (2015), pp. 820–841 (cit. pp. 447, 450).

[36] Damir Nesic, Jacob Krueger, Stefan Stanciulescu, and Thorsten Berger. “Principles of feature modeling”. In: *FSE*. 2019 (cit. p. 440).

[37] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. “A study of feature scattering in the Linux kernel”. In: *IEEE Transactions on Software Engineering* 47.1 (2021), pp. 146–164 (cit. p. 437).

[38] pure-systems GmbH. “pure::variants Eclipse Plugin User Guide”. 2004. URL: https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf (cit. p. 439).

[39] Matthias Riebisch. “Towards a more precise definition of feature models – position paper”. In: *Modelling Variability for Object-Oriented Product Lines*. Ed. by Matthias Riebisch and Detlef Streitferdt James O. Coplien. BookOnDemand Publ. Co., 2003, pp. 64–76 (cit. p. 438).

[40] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. “Cloned product variants: From ad-hoc to managed software product lines”. In: *STTT* 17.5 (2015), pp. 627–646 (cit. p. 445).

[41] Klaus Schmid. “Scoping software product lines”. In: *Software Product Lines*. Springer, 2000, pp. 513–532 (cit. p. 446).

[42] Daniel Strueber, Mukelabai Mukelabai, Jacob Krueger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. “Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems”. In: *23rd International Systems and Software Product Line Conference (SPLC)*. 2019 (cit. p. 442).

[43] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. “Featureide: an extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79 (2014), pp. 70–85 (cit. p. 439).

The features in the feature model will be used to generalize and parameterize other models.

Kyo Kang et al. [36]

13 Model and Language Variability

In the last two chapters, we discussed the use of MDSE techniques for realizing software product lines. More specifically, we described the realization of variability in traditionally developed systems and focused on variability of source code to customize it to particular needs. Let us now discuss the other direction: using product line techniques to reuse models and DSLs. Since engineering a DSL and its infrastructure (e.g., editors, analyzers, interpreters) can be expensive, you might want to reuse both. Like for source code, we can foster reuse by realizing variability in models and languages.

We already learned that the automotive domain has massive variability (Sect. 11.3). Increasingly often, automotive systems follow the AUTOSAR standard [21, 59, 65], which is a component framework, an operating system, and a set of modeling languages for developing software components that run in the electronic control units (ECUs) of modern cars. AUTOSAR improves the interoperability and reuse of software components across automotive suppliers and manufacturers. To handle the variation, elements in AUTOSAR are configurable and can be annotated with presence conditions (cf. Def. 11.4). Closely related to AUTOSAR is the *EAST-ADL*, an architecture description DSL specifically targeting automotive embedded systems [16]. EAST-ADL focuses on a higher level of abstraction than AUTOSAR, but reuses its entities for modeling the lower levels. EAST-ADL allows model elements to be annotated with presence conditions and offers feature-modeling capabilities—recognizing the vast need for variability management in automotive systems. Also other common modeling languages such as UML diagrams (e.g., use case diagrams [26] and state charts [43]) and Petri Nets [52] have been extended with variability-modeling capabilities, allowing variability to be represented in the respective models.

13.1 Case Study: Variability in our FSM DSL and its Models

Let us take a look at our FSM (finite-state-machines) DSL and FSM models again. It will allow us to nicely illustrate the different use cases of adding variability to models, which are concrete state machines in this case, such as our coffee machine model. Assume we need to maintain different variants of our coffee machine for different customers. There should be a variant that only supports brewing tea, one that only supports brewing coffee, one that supports both, and all these variants should sometimes come with a failsafe mode. That gives six variants, which you could still manage using clone & own. You could just have different copies of your full model, where

A. Wąsowski, T. Berger, *Domain-Specific Languages*, https://doi.org/10.1007/978-3-031-23669-3_13

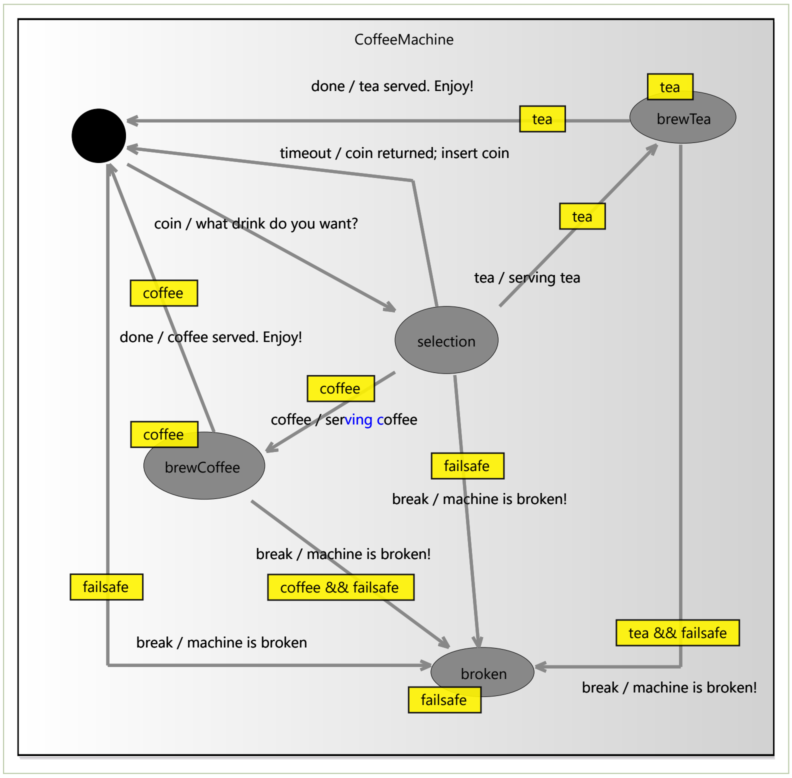

CoffeeMachine
done / tea served. Enjoy!
tea
tea
brewTea
timeout / coin returned; insert coin
coin / what drink do you want?
tea
tea / serving tea
coffee
done / coffee served. Enjoy!
selection
coffee
coffee / serving coffee
coffee
brewCoffee
failsafe
break / machine is broken!
break / machine is broken!
failsafe
coffee && failsafe
tea && failsafe
break / machine is broken
broken
failsafe
break / machine is broken!

Figure 13.1: Our coffee machine state machine with variability (presence conditions are annotations, but here visualized as yellow notes)

you removed the unnecessary parts. You could also enhance clone & own a bit by referencing other model elements to reduce redundancy. You could define one model as the explicit base model and then have other models that refer to contents from the base model. In our scenarios, this would totally do and probably be the easiest solution. But of course, while clone & own is simple, you run into the typical scalability problems when you have more variants and you need to maintain them (e.g., make modifications to parts that exist in multiple variants). So, let us instead realize what is commonly known as a *model template*, a *150 % model*, or a *model product line*.

Figure 13.1 shows our coffee machine model with variability represented as presence conditions (in the yellow notes) over three Boolean features: coffee, tea, and failsafe. Observe these conditions, which are attached to transitions and to states. For instance, the state brewCoffee and its incoming and outgoing transitions are only present when the feature coffee is enabled. The conditions are similar for the feature tea. Further observe that the transitions to the state broken from the optional states brewCoffee and brewTea can only be present when both failsafe and coffee, or when both failsafe and tea, are enabled, to avoid dangling transitions.

The presence of features is decided through a configuration at design or build time, not at runtime. So, the yellow annotations represent, for instance, the choice that the machine supports brewing tea. This choice can be made by the coffee machine manufacturer before a sale or before deploying it at the customer's site. The variable tea (input, i.e., trigger, of the transition from state selection to state brewTea), on the other hand, represents a runtime condition, specifically, that a user just requested to brew tea. This functionality is only available when the feature tea was enabled before, of course.

Now assume that the coffee machine is larger. In such a scenario, the manufacturer would likely, in addition, model the features in a feature model and implement a model transformation that generates code from the state-machine model. Here, it is important that the transformation includes a configuration step, so the transformation needs to be variability-aware and to evaluate the presence conditions in order to obtain a configured state-machine model (so, an M2M transformation) or directly generate code that omits states and transitions not enabled in the specific configuration (so, an M2T transformation). It is also clear that the graphical editor needs to support these yellow annotation boxes. As we will see below, the support for variability can be built into the DSL from the very beginning, so the developer of the concrete syntax can provide support, as well as the developer of the transformations. Another option is to inject the variability support from the outside and retroactively make the DSLs variability-aware, including its infrastructure (e.g., editors, analyzers, transformations). An example of automatically extending DSLs and their graphical, Sirius-based editors is provided by Garmendia et al. [22].

After briefly discussing variability in FSM models, let us see whether it might make sense to have variability in the FSM language as well. Assume we want to maintain different variants of the language: some variants with support for modeling hierarchical state machines, some variants that do not allow an output, or perhaps others that support timed transitions or expressions on the transitions. This would account for different language users and usage scenarios of state machines. Similarly to our coffee machine example, we could of course manage this small number of variants using clone & own, which has the advantages and disadvantages we discussed above. As opposed to a 150 % model, however, one usually needs to maintain copies of the language infrastructure (e.g., editor, transformations) as well. In other words, you duplicate abstract syntax, concrete syntax, and semantics. To avoid clone & own, we will realize a so-called language product line for our FSM language.

Figure 13.2 shows our FSM meta-model with two presence conditions attached to meta-model elements. The yellow notes are meant to illustrate Ecore annotations, which are usually not visible in an Ecore editor, but we added them here manually as notes. Ecore annotations are a powerful, but not well-documented part of Ecore. Basically, one can attach an annotation to almost any Ecore model element, and an annotation can be anything from text to more complex Ecore elements. OCL constraints are usually added

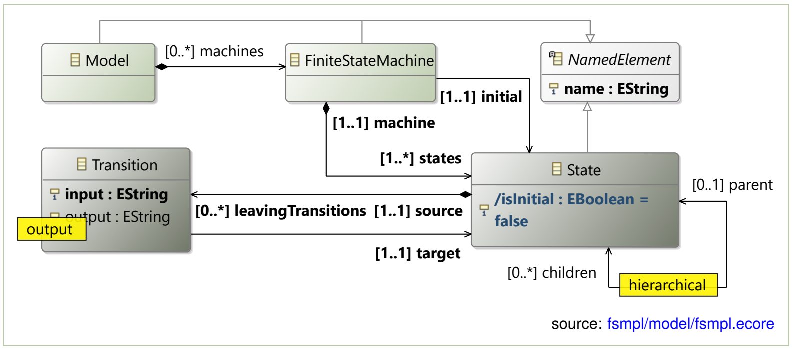


Figure 13.2: Our state-machine language (FSM) with variability (presence conditions as annotations in yellow boxes)

to Ecore models using Ecore annotations. Open our FSM meta-model that has those annotations (fsmpl/model/fsmpl.ecore) in Eclipse.

We illustrate the realization of two optional features: hierarchical and output. The feature hierarchical enables our states to be hierarchical. The feature's presence implies the presence of an association realizing a tree structure among states. The feature output enables the state machine to output text when a transition is taken. So, both features are useful, but not mandatory. Of course, in practice, it might not be so necessary to disable output; one could just not use it in models, but it is apparent that for larger languages and more complex features (e.g., those that are cross-cutting), tailoring a language will reduce complexity and not expose features irrelevant for the current use case or customer.

Let us build such a variability support for our FSM models (i.e., the meta-model as in Fig. 13.2 and its instances as in Fig. 13.1) to realize the variability above. Since we attach presence conditions to model elements, we need a language to represent such expressions over features as well as concrete configurations. Figure 13.3 shows a simple language for expressions and their configurations (i.e., assignments of literals to Boolean values). Note that we reuse the class Literal to also store their values (the attribute value is optional, so we set its cardinality to 0..1) in a configuration, so Literal has two roles. It represents the literals in expressions, where they do not have a value (the attribute is absent). It also represents literals in a configuration, where they have a value (of type Boolean in our case).

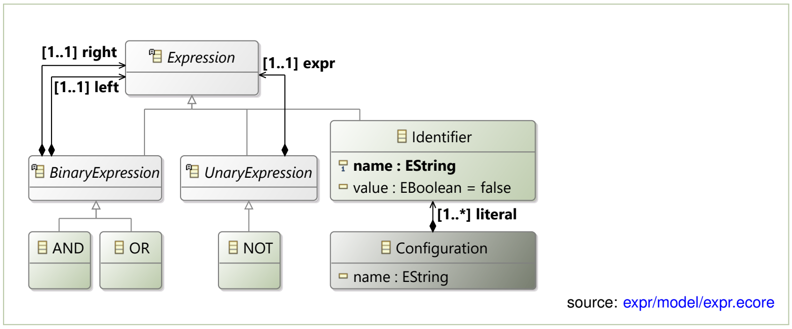


Figure 13.3: A simple expression language

```
grammar dsldesign.expr.xtext.Expr with org.eclipse.xtext.common.Terminals

import "http://www.dsl.design/dsldesign.expr"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Expression:
    Conjunction ( {OR.left=current} "||" right=Conjunction)* ;

Conjunction returns Expression:
    Unary ( {AND.left=current} "&&" right=Unary)* ;

Unary returns Expression:
    '(' Expression ')' | {NOT} "!" expr=Unary | Identifier;

Identifier returns Identifier:
    name=EString;

EString returns ecore::EString:
    STRING | ID;
```

source: expr.xtext/src/main/java/dsldesign/expr/xtext/Expr.xtext

Figure 13.4: *The grammar for the simple expression language for representing presence conditions of model elements*

To attach such expressions to elements in our Ecore meta-model, we use annotations (EAnnotation). These could in principle hold any structured meta-classes, so we could import the expression meta-model and then instantiate it in annotations, but this way of using EAnnotations is rare and has several disadvantages. Instead, we attach the presence conditions as expressions in textual concrete syntax. Eclipse's Ecore editor allows doing so in the properties view. Specifically, one attaches annotations as key-value pairs. In our case, we choose "condition" as the key and the textual expression as the value. This allows quite easy use of our simple expression language for models in languages other than Ecore as well. We define the syntax in Xtext, following what we learned in Chapter 4. It is shown in Fig. 13.4. Observe how we avoided a left-recursive grammar.

Next, we need an evaluator for expressions based on a configuration as well as some utility methods for creating a configuration. Even though we used Ecore and Xtext, where staying in the EMF world would be a natural choice, we implemented these methods in Scala. Furthermore, we could instead have expressed the FSM meta-model using algebraic data types instead of Ecore (e.g., as Scala case classes) and we could have written the parser directly in Scala using combinator parsing. However, we decided to show this combination to illustrate how Ecore models can be accessed from Scala—in addition to our model-transformation examples in Sect. 7.2.

Figure 13.5 shows our utility methods. It makes sense to also look at the meta-model of our expression language (Fig. 13.3) when you try to understand the code. First, `prettyPrint()` pretty-prints expressions. It is a simple recursive method that uses pattern matching to distinguish the nodes it is traversing, but since we are not using Scala case classes, we need to call the respective methods (e.g., `getLeft()`) to obtain the sub-nodes into which to descend further. We can only pattern match on the type of the node in the AST of the expression at hand. Second, `printConfiguration()` iterates through the literals contained in the configuration and prints their value. Third, `createConfiguration()` is owed to the relatively complex way of instantiating Ecore classes, relying on its factory methods and the enforcement of getters and setters, and dedicated methods to add values to lists. If we had used Scala case classes, it would have been much easier to realize instantiating configurations. Finally, `eval()` evaluates expressions for a specific configuration. Similarly to the pretty-printer, it recursively descends the expression AST with limited pattern matching (only on the type), and evaluates the nodes according to their semantics. Our code repository contains code illustrating how to invoke these methods and parse expressions in textual syntax (expr.xtext.scala/src/main/scala/dsldesign/expr/xtext/scala/xtextParserExampleMain.scala).

```scala
def prettyPrint (e: Expression): String =
  e match
    case a: AND =>
      s"(${prettyPrint (a.getLeft)} && ${prettyPrint (a.getRight)})"
    case o: OR =>
      s"(${prettyPrint (o.getLeft)} || ${prettyPrint (o.getRight)})"
    case i: Identifier => i.getName
    case n: NOT => "!" + prettyPrint (n.getExpr)
    case _ => "[unknown expression node]"

def printConfiguration (c: Configuration): String =
  c.getName + ": " + c.getLiteral.asScala.map( l =>
    l.getName + "=" + l.isValue ).mkString(", ")

def createConfiguration (name: String, c: List[(String, Boolean)])
  : Configuration =
  val eFactory = ExprFactory.eINSTANCE
  val result = eFactory.createConfiguration
  result.setName (name)
  for (n,v) <- c do
    val i = eFactory.createIdentifier
    i.setName (n)
    i.setValue (v)
    result.getLiteral.add (i)
  result

def eval (e: Expression, c: Configuration): Boolean = e match
  case a: AND =>
    eval (a.getLeft, c) && eval (a.getRight, c)
  case o: OR =>
    eval (o.getLeft, c) || eval (o.getRight, c)
  case i: Identifier =>
    c.getLiteral.asScala
      .find { _.getName == i.getName }
      .exists { _.isValue }
  case n: NOT =>
    eval (n.getExpr, c)
  case _ => false
```

source: expr.xtext.scala/src/main/scala/dsldesign/expr/xtext/scala/package-exprparserutils.scala

Figure 13.5: *Scala methods to pretty-print expressions, print configurations, instantiate configurations, and evaluate configurations*

Finally, let us implement the preprocessor that will derive models. It takes a configuration and a model with variability—models where elements are annotated with presence conditions—and outputs a model containing only elements without a presence condition or where the condition evaluates to true. In principle, other configuration mechanisms could be incorporated as well, such as replacing symbols representing features with the exact values. One could also weave in model elements from other models, which would be a more compositional mechanism (explained shortly), instead of the annotative one we realized here. Figure 13.6 shows our implementation in Scala. While it is pragmatic, basically just traversing all model elements and either just removing the annotation (when its presence condition evaluates to true) or removing the annotated model element, it is not very scalable for large models. In Ecore, removing elements triggers some expensive model traversals, so a more efficient implementation would collect all elements to remove and then remove them in one go. In our code repository we provide an example of how to apply this model configurator to our state-machine meta-model from Fig. 13.2 (in the `main` method in modelconfig/src/main/scala/dsldesign/modelconfig/scala/package.scala).

```scala
def deriveModel (epackage: EPackage, conf: Configuration): Unit =
  val c = EcoreUtil.getAllProperContents[EObject] (epackage, false)
  for item <- c.asScala
  do item match
    case e: EModelElement =>
       e.getEAnnotations.asScala
         .find { _.getDetails.containsKey ("condition") }
         .foreach { annotation => // at most one
           val cond = annotation.getDetails.get ("condition")
           if evaluateCondition (cond, conf)
             // include element, i.e., just remove annotation
             then EcoreUtil.delete (annotation)
             // do not include element, i.e., remove the element
             else EcoreUtil.delete (e)
         }
    case _ =>

private lazy val setup = ExprStandaloneSetup ()

def evaluateCondition (cond: String, conf: Configuration): Boolean =
  given org.eclipse.xtext.ISetup = setup
  eval (cond.parse[Expression].get, conf) // crash on a parse error
```

source: modelconfig.scala/src/main/scala/dsldesign/modelconfig/scala/package.scala

Figure 13.6: Ecore model configurator

13.2 Benefits of Variability in Models and DSLs

The most general benefit of variability in DSLs is that it enables reuse not only of the DSL, but also of its infrastructure, both of which can be difficult to develop.

Language product lines allow end-users to customize DSLs to specific domains without needing to be trained in language design [9, 34]. While

MDSE promises to let developers who are not experts in language design develop custom DSLs, having variability in them will even allow non-developers (e.g., domain experts or end-users) to create their own customized DSLs. This is a middle ground between offering an off-the-shelf DSL and letting developers create DSLs from scratch.

Compared to clone & own, the core advantage is that variability avoids redundancy when maintaining multiple model and language variants together with their infrastructure. Of course, one could enhance clone & own by defining a base model, which the variant models could refer to via referencing, which reduces redundancy, but that does not really avoid the scalability problems of clone & own.

When adding variability to models, you will likely end up with fewer variation points than when you add them to code. The intuition is that DSLs and models are defined at higher levels of abstraction, so it is safe to assume that features can be more modularized and are less cross-cutting. When concrete (meta-) models are derived—through a feature-model configuration and a model transformation—the more detailed code is generated in a systematic way. The higher abstraction especially allows modular representation of cross-cutting concerns which could unfold into cross-cutting code. The disadvantage is that you not only need to define DSLs, but you also need ways to express variation points in them, which can be done in different ways and at different levels of intrusiveness.

Let us also discuss the benefits of the pragmatic way of realizing variability in languages and models that we described in our case study above. The core advantage is that it can be applied to any Ecore model, since we use an annotation mechanism built into the Ecore language. This way, we did not even have to extend the visual editors—neither the visual class diagram editor for the meta-model nor the visual editor for FSM models. Both by default show a properties view that can be used to add those annotations. Of course, if we wanted to have visual editors that show the annotated presence condition as yellow boxes as in Fig. 13.1 and Fig. 13.2, we would need to extend the editors. There are techniques to generate the latter, however [22]; or with projectional workbenches, much more comprehensive support is possible—in prior work we developed a variability language that offers colored bars to specify presence conditions when composed with any programming language available in Jetbrains MPS [5, 51].

In addition, our preprocessor needs to be added to the transformation chain, or alternatively existing transformations need to be extended.

13.3 Variability Mechanisms for Models

As an alternative to managing model variants using clone & own, one can also manage them by integrating these variants into a platform and deriving individual variants from the platform. To this end, a platform contains variation points realized using specific implementation techniques called variability mechanisms. This strategy is similar to creating a platform for software variants, but there are some specifics for models.

Most importantly, models and languages usually do not stand alone, but come with an infrastructure, including visual or textual editors and transformations. In other words, when adding variability to models or languages, you need to consider three dimensions—abstract syntax, concrete syntax, and semantics [1, 9]. Especially when models play the role of a meta-model for a language, the infrastructure comprises an editor for the concrete syntax and transformations, generators, or interpreters for the semantics. So, variability also affects this infrastructure. Since abstract syntax, concrete syntax, and semantics are represented differently, they might require different variability mechanisms. Our FSM example above illustrated this core challenge of variability in models, where we declared the abstract syntax (meta-model) in the Ecore language and the concrete syntax in an Xtext grammar. We did not consider the semantics of the FSM language in our example, but we defined the semantics of variability annotations in the FSM meta-model and in FSM models within our model configurator tool.

We call a model with variability a *model product line*. When the model is the meta-model of a language, then we call it a *meta-model product line* or a *language product line*. Their variability mechanisms are classified into *annotative* and *compositional* mechanisms [5, 37], similarly to variability mechanisms for source code. For adding variability to models, we also distinguish between the *amalgamated* and the *separated* strategy.

We will discuss different variability mechanisms for models in the remainder. Among other things, we will see that there is a lot of support for realizing variability in the abstract syntax, while there is not so much support for the concrete syntax (and for model editors) and the semantics, which can be tricky to realize.

Research has focused on providing languages and tooling for adding variability to DSLs, as opposed to ad hoc solutions. Let us call such a language a variability language.[1] Various frameworks for realizing variability in models have been developed, which we will also briefly mention. According to our experience, they are all somewhat difficult to set up, so you might be better off creating your own solution based on our FSM example in Sect. 13.1 above and the engineering process we propose in Sect. 13.4 below.

Annotative versus Compositional Variability

Annotative variability. Our FSM example above was annotative. Annotative (a.k.a. negative) variability relies on integrating all the variability into one model. Such a model is made configurable by means of annotations,

[1] A variability language focuses on the solution space and is different from a variability-modeling language (e.g., feature modeling), which focuses on the problem space. They are mapped to each other.

which specify how the model should vary. We specifically call such a model a *150 % model* or a *model template*, which is a model product line with annotative variability. A concrete variant is derived from the 150 % model by binding variation points as defined in the annotations.

Realizing annotative variability in models requires an annotation language to express annotations that realize variation points, and it requires tooling that derives concrete model variants. Annotation language is a more specific term for a variability language in annotative variability. In a very simple annotation language, as in our FSM example above, derivation of a concrete variant from the annotated model amounts to removing the parts that are not enabled by the configuration. This can be achieved by the annotations containing presence conditions, which are expressions over features (cf. Def. 11.4). If they evaluate to true for configuration then the annotated model part remains in the concrete variant model. Beyond annotations just specifying the presence or absence of elements, annotations can also be more expressive and describe variation points that not only control parameterization and existence, but also substitution and user-defined variability. They can also be more domain-specific.

The most mature framework for annotative model and language variability is probably Base Variability Resolution (BVR) [27, 28, 71, 29], also known as Common Variability Language (CVL) [30]. It provides a full variability management solution for models, including feature-modeling capabilities and an expressive annotation language that allows specification of expressive variation points. For instance, variation points can be of type existence, substitution, or value assignment, among others. The BVR/CVL framework allows variation points to be added to any MOF-compliant model, called the base model. Its main implementation [71] is built on top of EMF, so it supports any Ecore model. In BVR/CVL, variation points are controlled and mapped to features[2] in a feature model.[3] Other frameworks are FeatureMapper [33] and the more recent SuperMod [60, 45].

Annotation languages can not only be more expressive, but also be more domain-specific. For instance, we could have an annotation language for our FSM models where we can describe in domain terms (e.g., states, transitions) that a state should connect to a specific other state when a certain feature is enabled. This would also allow topological variability to be realized [6], where a model varies based on how model parts are connected, as you recall from our fire alarm system case study in Sect. 11.5. With our simple annotation language based on presence conditions, realizing this variation would potentially lead to many redundancies and a 150 % model that would be difficult to comprehend. A dedicated annotation language[4] that allows description of such variations in domain terms could make describing such variation much easier. In fact, our fire alarm system

[2]Called VSpec in BVR/CVL.

[3]Called the VSpec tree in BVR/CVL.

[4]In fact, calling it an annotation DSL would be more appropriate and emphasize the domain-orientation.

meta-model could be seen as an extreme case of such a domain-specific annotation language intermingled (amalgamated, see below) with common domain concepts we used to model the fire alarm system domain.

There are techniques to develop expressive and domain-specific annotation languages. One such technique is VML* [82], which is a family of languages that allow construction of DSLs that describe how models vary. VML* provides various actions that will perform the variation, including removing and replacing model parts, but the abstractions presented to the developer in an annotation language built using VML* can look very different (so, they abstract over removing and replacing model parts). Alternatively, Sánchez et al. [57] and Greifenberg et al. [23] provide engineering processes for such languages.

The variability mechanisms above all focus on the abstract syntax. For concrete syntax and semantics, we do not get much support from the existing literature, which primarily focuses on the abstract syntax for annotative variability. However, supporting the concrete syntax is important to allow an editing infrastructure that can deal with variability annotations in models. Beyond resorting to textual annotations, which are offered by typical textual preprocessors (e.g., C preprocessor, Antenna, Velocity, or PHP), few works provide support. One of the few works that help you add variability to visual DSLs is the work by Garmendia et al. [22], which takes a visual DSL and generates a variability-aware visual DSL. The respective editor allows annotation of models in the DSL with presence conditions and looks similar to our example in Fig. 13.2. For compositional variability, there is some more support, as we discuss shortly.

For the semantics, which are often implemented as generators, model transformations, or interpreters, one can resort to the facilities (e.g., IF statements) of a GPL or a model-transformation language. Salay et al. [56] provide a technique to lift model transformations to variability. We will also discuss very pragmatic ways of realizing variability in concrete syntax and semantics in Sect. 13.4.

Compositional variability. Alternatively, *compositional (a.k.a. positive) variability* advocates decomposing variable model parts into separate models, instead of integrating the variability in one model. The separate models are usually called composition units, feature modules, delta modules, or just modules [4, 3, 38, 58]. Concrete variants are obtained by composing modules according to a concrete configuration. Often, a base model that contains the commonality and separate models for optional features are created.

A compositional variability mechanism comprises a variability language for expressing model fragments (called modules) and a model composition technique. In other words, the meta-model will have facilities to describe modules, including for instance their interface and the position(s) in a base model into which they should be composed.

Many DSLs already provide simple file inclusion mechanisms, which can be used to modularize variable parts. However, the variability encapsulated in modules is often cross-cutting and needs to change other parts. There are often also constraints among modules to avoid unwanted interactions or to satisfy dependencies. So, more expressive mechanisms are needed.

Many compositional variability mechanisms rely on model composition techniques that have been conceived for common use cases beyond just variability. In the literature, such support exists mainly for the abstract syntax (e.g., meta-model) and the concrete syntax (e.g., grammar), partially also for the semantics (e.g., model transformation). For the latter, the main challenge lies in combining semantics implementations, which requires solving a viable execution order. This is a complex problem and requires declarative transformations that can be fed to a solver that will produce such an order for a selection of language modules to combine. A discussion about this problem is provided by Völter and Visser [78].

We distinguish between *reference-oriented* and *merge-oriented* techniques. For the former, there is a range of such techniques, including grammar and meta-model inheritance and importing techniques, which 'virtually' compose models. Of course, the referencing needs to be variable (e.g., import another model based on a presence condition), which can be achieved using variability annotations. The other techniques rely on merging the actual models, so, 'physically' merging them.

Let us illustrate some merge-oriented model composition techniques.

- *Model superimposition* is a syntax-oriented composition technique that merges modules (i.e., model fragments) based on their partonomies (the main hierarchical structure in models). The idea is to merge corresponding substructures based on nominal and structural similarity. The nodes in the hierarchy need to have a name (unique in the scope of the parent module). The comparison of these structures determines the merge point. Typically, one has a base model with a structure, then other models with a partial slice of this structure are merged into the base model. This high-level explanation might not be too intuitive, but in the end it is a relatively simple composition technique. Take a look at the descriptions and examples by Apel et al. [2], who present a language to express modules that can be composed into a concrete model variant using superimposition.

- *Aspect-oriented composition* relies on mechanisms known from aspect-oriented programming, where one has specific languages to express the positions (called join points) in target models where model fragment should be integrated. Morin et al. [50] present a technique to add variability concepts to an existing meta-model, allowing model fragments to be expressed and then woven together. The difference to superimposition is the way the target location in models for integrating model fragments is described. The specific technique of Morin et al. [50] calls model fragments graft models and the target the base model. It supports adding model elements, modifying properties of model elements, and merging model elements. More aspect-oriented composition techniques for models exist [42].

- *Three-way merge* is a common file composition mechanism in version-control systems. Traditionally, three-way merge integrates evolutionary changes stemming from concurrent development, but it can also be used to merge modules. For realizing variability in models, the selection of modules to be merged is controlled by a configuration of features. So, features are realized by modules, which are selectively merged to obtain a concrete variant. Since three-way merge requires three files, a core limitation is that modules need to be organized in a certain way.
- *Custom merge* means that merging is often domain-specific, and it can be implemented individually. There is support for custom merge. For instance, the generic "weaving tool" ATLAS model weaver [8] allows definition and generation of specific merge tooling. Conceptually, when two models should be merged (each of which conforms to a meta-model), the idea is that the developer creates a weaving meta-model, an instance of which defines relations between elements of the models, and which controls the merge.

Frameworks for compositional model variability rely conceptually on the composition mechanisms above, but provide more specific variability languages to define modules and their possible compositions. They address the problem that modules can have dependencies and might interact in certain ways, which needs to be managed. Among other things, derivation of invalid languages must be prevented. To this end, the frameworks add such mechanisms, most often by relying on feature models to define the available modules and their dependencies. The languages to define modules can also be more domain-specific—to the domain of variability or the target domain of the model.

On the language level, there are component-oriented language workbenches that more naturally support reuse and combination of language modules. They rely conceptually on model composition techniques, but of course need to support the language infrastructure as well. Examples of component-oriented language workbenches are Jetbrains Meta Programming System (MPS) [75, 7], Neverlang [67], MontiCore [39, 9], LISA [49], and JastAdd [32]. The idea is to enhance reuse beyond what more mainstream language workbenches offer, including Xtext. For language-level frameworks that support variability, the same motivation holds as for model composition with variability support by means of feature models. The language modules have dependencies and often need to be composed in certain ways. For example, Neverlang has been connected to BVR/CVL [68]. The motivation is that modules (or parts of modules, such as the concrete syntax definition or the semantics) have dependencies, including requires and excludes, which might be difficult to manage for many language modules. Modules might have interactions. So, it is difficult for developers to adhere to and manage such constraints. Later, Neverlang has also been connected to FeatureIDE [20]. Jetbrains MPS has dedicated support for feature models.

Despite all the support, composing languages is still a challenge. The support is often limited to one technological space (e.g., Eclipse EMF) and lacks common foundations. Also recall that languages consist of abstract syntax, concrete syntax, and semantics, each of which can require different mechanisms. Especially for the concrete syntax, composing grammars can be an intricate problem, since grammar ambiguities might arise, which need to be resolved. We refer to the literature for more information [34, 69], especially the literature on projectional language workbenches such as MPS, which eases the combination of concrete syntaxes of language modules and avoids the problem of grammar ambiguities [7, 75, 19, 78, 76].

Amalgamated versus Separated Variability

Recall that both annotative and compositional variability mechanisms need a variability language to describe variation points (annotative variability) or the composition of modules (compositional variability). For annotative variability, the variability language can be a combination of annotations and the expression language for our FSM example above, since it makes it possible to express that a certain model element is present or not present based on a certain configuration for which the presence condition expression evaluates to true. In our example, the annotations were already part of the meta-modeling language Ecore, while we introduced the expression language. This strategy is called *amalgamated* or *intrusive*—your meta-model has elements to describe variability [28]. Alternatively, the variability concepts can be defined completely outside your DSL, which is called *separated* or *non-intrusive* variability.

In *amalgamated variability* you have the concepts from the variability language (e.g., variation points or annotations) directly in the language. Either the language already offers facilities that can be used, or you need to define them. Languages that offer such facilities are, for instance, UML, which has stereotypes, tagged values, and structural constraints; or Ecore, which has EAnnotations, as we used them in the FSM case study above. When we use such facilities in this way, we call them *ad hoc variability extensions*. If they are a bit more formalized and reusable, we call them *ad hoc variability languages*. Examples are UML profiles offering variability concepts [80, 81]. There are also *generic variability languages* that can be used for any language in a certain technological space (e.g., EMF or Jetbrains MPS). Examples are our variability language in PEoPL [5], VML* [82], and the variability language offered by Garmendia et al. [22].

In amalgamated variability, your models will then use the domain-specific concepts of the actual language and the variability concepts available via one of the three strategies—ad hoc variability extensions, ad hoc variability languages, or generic variability languages.

Symbol	Description
❓	Decision affecting following activities
⚙	Activity
(⚙)	Optional activity
⚙⚙	Composite activity
(⚙⚙)	Optional composite activity
⚙	Sub-Activity of a composite activity

Table 13.1: Legend for designing model and language product lines

There can also be good reasons for not wanting to extend meta-models with variability description facilities—for instance, when you do not want to modify existing tooling. Then, a *separated variability* or *non-intrusive* strategy makes sense [28]. This allows description of variation points and model composition completely outside the model in a separate model. A variability language supporting this strategy needs to have a way to point into existing models in a non-intrusive way. The framework BVR/CVL [71, 29, 30] we mentioned above provides this support.

13.4 Designing Language Product Lines

Languages like our FSM language with variability are called *meta-model product lines* or *language product lines* [24, 41, 25]. As you can see from these examples, the mechanism at both levels—the language (a.k.a. meta-model or M2) level and the model (a.k.a. instance or M1) level—is the same. We attached presence conditions to the elements of models, which are then called *model templates*, *150 % models* or *model product lines* [14, 9, 33, 2, 29]. Since the technique is the same, we will talk about language variability, but mean model variability as well (the intuition is similar to our discussion in the box on page 396).

As we already indicated above, when discussing variability mechanisms for models, the design space for realizing variability in languages is huge. Like for traditional product lines, there is a huge body of knowledge stemming from research on model variability. This is not surprising, since the domains where SPLE is thriving are usually model-oriented—recall automotive, avionics, and industrial automation. Let us more systematically discuss the design space and what kinds of decisions you will need to make to realize variability.

We now describe a development process and discuss pragmatic solutions in addition to using some of the existing frameworks for realizing variability we mentioned above in Sect. 13.3.

⚙ *Domain analysis.* As with traditional product lines, it is very helpful to have an understanding of the variability you think you will encounter in the future. The value of this activity should not be underrated [79] for DSLs. A domain analysis together with a scoping process, where you come up with features that you organize in a feature model, will help. You can also attach priorities to those features for planning. The features you will need arise from understanding the users of your models (or languages) and the anticipated usage scenarios. You should also think about the complexity and scale of your models.

In principle, you can build language product lines bottom-up or top-down. In the latter case, you would do the domain analysis after the other activities and identify features by diffing the variants, possibly using our feature-modeling process from Chapter 12. However, in practice, we believe that distinguishing between top-down and bottom-up will hardly make a difference to the number of variants and their sizes. We advocate being pragmatic here. Identify the features and think about future features, and consider the following factors, but do not limit yourself to a strict process or order of activities. Whether language variants will ever be as numerous as variants of software systems remains to be seen.

Variants needed? The domain analysis will help you decide whether variability is really necessary or a one-size-fits-all language will suffice. Tailoring the languages with variability will make them easier to use, but with the tradeoff of having to manage the variants. That is a difficult decision you will need to make. If you cannot make it, it makes most sense to avoid variants and only incorporate them later when necessary—which is also the common adoption strategy for traditional product lines. We showed in our FSM case study how to add variability to languages and models retroactively.

Decision binding time? Another important question is about *when* to make decisions. The system you are describing will have some functional logic, allowing its operation at runtime. In our FSM coffee machine example (Fig. 13.1), that is for instance the variable coffee, which represents an event at runtime. So, coffee will be true when the machine is in the state selection and a user requests coffee. Then, this decision will determine the behavior of our coffee machine. In other words, the decision what to brew is made at runtime. You might also decide that this decision should be made at design time, where the manufacturer decides what behavior to ship. In that case, there would be a variant with fixed behavior determined by the decision, which cannot be changed at runtime by a user. It is a conscious decision, which is often not as easy to make as for our coffee machine example, where it is clear from domain knowledge that the user should make the decision. In more complex systems, it can be difficult to decide. So, in summary, you will need to decide what should be part of the functional, control logic of the system and what should be part of the engineering process. In our coffee machine example, we decided that the presence of the functionality to brew coffee should be decided at build time by the manufacturer—this was also easy to decide based on domain knowledge (you need to decide about the presence of coffee-brewing hardware already on the manufacturer's side).

Clone & Own versus variability? When you decide you need to have variants (as opposed to putting everything into the runtime), the question arises whether to go for clone & own versus adding variability. The decision is based on the number of variants and the extent of customization of the surrounding language infrastructure, since variability can affect abstract syntax, concrete syntax, and semantics. Recall that when you need a custom editor for each variant, you need to maintain those copies as well; similarly for transformations. Also recall that you can enhance clone & own by defining a base model from which the variant models can reuse elements via referencing, which is a middle ground between clone & own and variability [66]. So, anticipate the effort needed to maintain those parts when making a decision. Researchers have tried to help you with such a decision, but only some rough guidelines are available, such as the rule of three variants determining the break-even point when a product line pays off [70, 54], or recent empirical data on the costs of either strategy [40].

? *Annotative versus compositional variability?* When you decide to realize your variants through variability mechanisms, the next question is what kind of mechanism to choose.

In practice, annotative variability is more common (e.g., recall our Linux kernel case study from Sect. 11.2) and typically preferred by developers as the easier technique [64, 12, 37, 46]. Models and other artifacts can be annotated in a very fine-grained manner. A core challenge is that annotations easily clutter the models and other artifacts, challenging their comprehension.

Modularity fosters comprehension, since modules can be developed and maintained separately, which is less intellectually challenging. However, compositional mechanisms lead to more coarse-grained variability compared to annotations, and the effort to create modules is typically higher, since you need to find a good decomposition of your model variants into modules.

Compositional variability offers better support for the concrete syntax and the semantics, in addition to the abstract syntax. Compositional language workbenches (cf. Sect. 13.3) offer modules that encapsulate the three parts, requiring developers to think about the modularization much more, and offering tool support for the three parts out of the box. When you have many variants, there are frameworks that extend these workbenches with facilities to manage the language modules better (e.g., Neverlang combined with FeatureIDE); see our discussion in Sect. 13.3.

In summary, this decision depends on the structure and complexity of your features, the realizability of a variability mechanism of either kind, the number of features and variants, and the granularity of your variability.

? *Amalgamated versus separated variability?* The next challenge is to choose or realize a language that you can use to describe variation points (annotative variability) or the composition of modules (compositional variability). You need to decide whether you put that language's concepts into the meta-model (amalgamated) or not (separated). Make the decision based on the following list of advantages and disadvantages.

Separated variability enhances comprehension of the language. Especially when you have annotative variability, it avoids the clutter of having annotations directly in the model. This comes at the cost of reasoning about variability and quickly distinguishing the variable from the common parts of the model. The latter usually needs tool support. Separated variability, based on how specific it is to the meta-model, might require co-evolution. However, non-domain-specific variability languages, such as in CVL/BVR, are only specific to the meta-modeling language (MOF/Ecore), so you won't have any problems there.

Amalgamated variability enhances comprehension of the variability that exists in the language. Especially when the language has extension mechanisms (cf. Sect. 13.3), that provides a huge benefit, since the tooling will be supportive in this case.

In summary, you will make this decision depending on the extensibility of your meta-model, your language infrastructure, and the comprehensibility of your DSL with or without variability concepts.

Select a technological space. Usually, you will extend an existing language with variability in a bottom-up way, so the language's implementation determines the technological space. However, if you build your language with variability support from scratch, in a top-down way, then you might want to select the space based on your decisions about the realization of variability. Especially if you decide for compositional variability, then it makes sense to select a compositional language framework, many of which we listed above in Sect. 13.3. In our opinion, the most mature and advanced compositional language framework is Jetbrains MPS, which we used successfully for various projects [44, 5, 76, 7, 77].

Beyond those compositional language frameworks, research has focused on providing standardized variability languages and tooling for language product lines. Various frameworks for realizing variability in models have been developed, many of which we also mentioned in Sect. 13.3. However, according to our experience, they are difficult to set up. So, selecting a technological space and a framework should be a careful decision.

Your decision whether you realize annotative or compositional variability further influences the choice of a technological space, and vice versa.

Realize annotative variability. Recall that variability in languages affects abstract syntax, concrete syntax, and semantics.

Parameterization Start with parameterization, which might be enough for your use case. You add support (either amalgamated or separated) for using placeholders (parameters) in your models. A preprocessor traversing the model and replacing the placeholders could be realized as a simple modification of our model configurator above (Fig. 13.6).

Annotations with Presence Conditions Then, check whether annotations with presence conditions are sufficient. They work well when you do not have too many feature interactions.[5] Annotations might be a bit cumbersome when you need to replace parts in the model with other parts. Then, a more expressive or domain-specific technique for describing variation points can be useful that not only controls parameterization and existence, but also substitution and user-defined variability (e.g., BVR/CVL and VML* in Sect. 13.3).

Annotative variability for the abstract syntax is relatively easy to realize, as we showed in Sect. 13.1. It gets tricky for the concrete syntax, since you would need to realize the same annotations for the concrete syntax definition (e.g., a grammar in Xtext or visualization rules in Sirius). In principle, textual annotations could be added to the concrete syntax using a textual preprocessor (e.g., C preprocessor, Antenna, Velocity, or PHP). A pragmatic solution is to provide the textual syntax for all variants. Then, depending on the derived meta-model variant (abstract-syntax definition), only the subset for the available meta-model elements will be available in the language infrastructure. However, this depends on the language workbench you use to realize the concrete syntax, where you will have to experiment a bit to figure it out. It will also not work when there is variability in the mapping (e.g., one concept in the concrete syntax maps to different concepts in the abstract syntax depending on the configuration). Finally, when your syntax definition relies on grammars, you can also leverage non-terminal symbols to represent variable parts—after all, that is a built-in variability mechanism in grammars.

For the semantics, similar issues arise as for the concrete syntax. Assume we implement the semantics using transformations. Then, these can also be made configurable. Fortunately, model-transformation languages are expressive (and are often GPLs, for instance, when we implement the transformation in Scala), so they naturally come with configuration mechanisms (e.g., IF statements) to realize variability. Alternatively, the superset of all transformation rules can be realized. Then, depending on the derived abstract syntax of models, only certain rules are used. So, you would implement the transformation against the product line meta-model (which is a 150 % model), instead of having different variants of transformations. Like for the concrete syntax, this works when the transformation logic is not affected by variability, such as when you need to interpret a source element differently based on the configuration.

Domain-Specific and Expressive Annotations Realizing more expressive or domain-specific annotations can be tricky. Of course, if annotations are similarly expressive to presence conditions—that is, they define presence or absence of meta-model elements, then they can easily be made domain-specific by using domain terms. When you need more expressiveness, for instance, for topological variability, you can resort to

[5] For feature interactions, it is often necessary to specify their handling, for which you need model parts that handle those. The presence conditions of these parts are then typically a conjunction over the interacting features.

a framework, but the added complexity might be huge. We experimented with CVL for topological variability and cannot recommend it. You might want to experiment with VML* or more recent frameworks. However, in the end, it might be quicker to develop your own DSL for it, either directly in your target meta-model (amalgamated) or outside (separated). This gives you full flexibility. For our FSM example (cf. Sect. 13.1), for instance, you could define a concept that has a presence condition as a parameter and in its body defines that a certain transition exists with specific source and target states. Our fire alarm system from Sect. 11.5 is a very expressive and very domain-specific variability language. However, it was not connected to feature models, which would allow a feature-oriented configuration.

⚙ *Realize compositional variability.* The idea of compositional variability techniques is to modularize your optional features into separate modules. You have one or multiple base modules that represent the mandatory parts of your models. This will allow you, with the help of a framework (either a compositional language workbench or a framework that adds variability-modeling capabilities with feature models on top), to compose the mandatory parts with the optional features based on a configuration.

Decomposing your language product line into modules is almost an art, but you want to follow some criteria:

- Decomposing is highly domain-specific, so you want to put modules together based on their distinctness in the domain. A module should correspond to a feature and as such be understandable for the target users of your language product line—developers who derive or customize languages.
- You want to have as few dependencies between modules as possible.
- You need to consider abstract syntax, concrete syntax, and semantics when decomposing.
- You might want to realize minor variations within or between modules using annotative variability, perhaps even just with parameterization (depending on what your framework supports).

Remember that some features are cross-cutting, so the parts of the model that belong to a module need to be carefully decided. Some frameworks (cf. Sect. 13.3) support cross-cutting features in modules, but not the compositional language workbenches. For the latter, you might need to introduce redundancy, and to control it, you might need to connect feature modules to features and then control the composition via a configuration.

It is also important to create specific feature-interaction modules, which handle wanted or unwanted interactions between features.

? *Feature model or configuration file?* If you use a framework that supports language variability with feature models, this question does not arise. When you create your variability mechanisms yourself, e.g., something similar to the annotative mechanism in our FSM example, the question arises where to specify the features. When you do not have many dependencies or when just documenting the dependencies in textual form, you could go with a simple configuration (a.k.a. properties) file. Put the dependencies into comments. Of course, the more features, dependencies, and variants you have, the more sense it makes to use a feature model and connect to a feature-modeling tool.

Quality assurance. We are not aware of any specific support for validating language product lines in the literature. You are largely on your own and should be pragmatic.

If you have built your language product line from existing variants, you should try to recreate the original variants. You might have test cases for the variants, which you might be able to reuse.

Otherwise, derive a set of variants based on your domain knowledge. You should try to create a minimal and a maximal language variant, but also combine features where you have the impression they could interact. For both strategies, you should instantiate the languages and run their semantics for validation.

Further Reading

Language composition. Language composition is a relevant concept for variability. In general, composition has many benefits, but from the human perspective, you can take a look at the following studies to see why developers seem to appreciate language composition: Hutchinson et al. [35], Selić [61], and Völter et al. [76].

Erdweg, Giarrusso, and Rendel [18] discuss language composition from a conceptual point of view and describe various kinds of composition together with examples. Specifically, they describe so-called language extension, language restriction, language unification, self-extension, and extension composition. Some of these are easier to realize (e.g., language extension is easier than composition), as discussed by Völter and Visser [78]. General thoughts on language composition and managing languages in the large are provided by Hölldobler, Rumpe, and Wortmann [34].

Combemale et al. [13] provide a chapter (Chapter 11) on model composition for the purpose of scaling up model management, where they discuss aspect-oriented weaving, reuse with typing, and slicing techniques.

Solutions to realize language and model product lines have been presented especially for the following four technological spaces.

Projectional language workbenches. The field of projectional language workbenches with its core technology, projectional editing (a.k.a. structured editing or syntax-directed editing), focuses on language composition and language modularity. We already discussed projectional workbenches as a special kind of language workbenches in Sect. 2.2. Going back to the 1980s, projectional editing is not even a new idea. Early projectional workbenches include the Incremental Programming Environment [47], GANDALF [53], and the Synthesizer Generator [55]. Projectional editing gained widespread attention with Charles Simonyi's paradigm of *intentional programming* in the 1990s [62, 15, 63, 11]. The currently most mature projectional editing workbench is Jetbrains MPS,[6] which we already mentioned above. Its main advantage for language composition relies on the absence of parsing. You can

[6] http://www.jetbrains.com/mps

develop language extensions modularly and embed languages into other languages, especially DSLs into GPLs [73]. Projectional editors avoid language ambiguities, because you always work directly on the AST. Your editing gestures change the AST directly, without any parsing. Since that AST is always maintained in memory, every node is always mapped to its defining language concept (meta-class). Another benefit is that visual notations can be used flexibly and mixed with each other and with textual notations. This includes notations that one cannot easily parse (e.g., tables, diagrams, and mathematical formulas) [74].

The book of Völter [72] focuses on building DSLs in MPS and discusses many of its capabilities in a very practice-oriented way. A book focusing on the technical details and covering MPS in depth and breadth is the one by Campagne [10]. Both books are must-reads for those who want to build languages in MPS.

In our own prior work, upon the projectional language workbench Jetbrains Meta Programming system (MPS), we developed variability support that can be easily plugged into programming languages [5, 51].

JastAdd as a compositional language workbench. JastAdd [32] is a compositional language workbench focusing on GPLs, but can of course also be used to develop DSLs. It supports creating full-fledged compilers, composed of language modules containing abstract syntax, concrete syntax, and semantics. A core feature is its support for attribute grammars—a programming technique to declaratively compute properties (i.e., attributes) of nodes in the AST to efficiently implement language semantics. An attribute could reference, for instance, the node defining a variable from its usage nodes. Attributes can depend on other attributes, and might change based on changes in the tree. Attribute grammars allow such attributes to be efficiently computed. Take a look at the tutorial about JastAdd by Hedin [31]. It is also interesting to read a modular implementation of a Java compiler using JastAdd—where the different language concepts added over time to Java can be added modularly via language module composition [17].

Other compositional language workbenches. More compositional language workbenches exist; many are not less extensive than the others above. Read about MontiCore, which focuses on DSLs, in Krahn, Rumpe, and Völkel [39], and about its extensions to further improve the reuse of language modules [9]. Read about Neverlang in Vacchi and Cazzola [67], which is even feature-oriented, that is, allows composition of languages based on selecting features that are mapped to modules.

Furthermore, compositional language workbenches have been proposed and extended with variability management concepts. Combemale et al. [13] provide a chapter on variability in models, which mainly explains CVL/BVR. Butting et al. [9] provide a framework for engineering language product lines. Méndez-Acuña et al. [48] discuss a bottom-up reverse-engineering process to obtain a language product line from existing language variants.

Additional Exercises

Exercise 13.1. Extend the Ecore model configurator from Fig. 13.6 so that it supports parameterization. Furthermore, assume you would use this preprocessor to parameterize a DSL. Describe this use case in more detail and explain how you would incorporate the preprocessor into the overall tool chain, including how you would define parameters and where you would store their values.

Exercise 13.2. Create a simple model weaver. For the weaving strategy, decide whether to use superimposition based on the model structure or another technique to merge model fragments. Read the respective literature on the details [2]. Create simple models to test your model weaver. As an extension, make the weaving configurable, i.e., attach presence conditions over features to the fragments, which the weaver will take into account. For that you can reuse our simple expression language and the model configurator, but you could also implement it in your favorite GPL. This assignment is rather advanced.

Exercise 13.3. Discuss properties that modules of languages should have. Describe what potential techniques for checking these properties could look like.

Exercise 13.4. Revisit our fire alarm system case study from Sect. 11.5. Identify three features (at least one should cross-cut multiple classes) and map them to classes and relationships. Choose and very briefly explain an appropriate mechanism to map the features.

References

[1] David Fernando Méndez Acuña. “Leveraging software product lines engineering in the construction of domain specific languages”. PhD thesis. Université Rennes 1, 2016 (cit. p. 467).

[2] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. “Model superimposition in software product lines”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2009, pp. 4–19 (cit. pp. 470, 473, 481).

[3] Sven Apel, Christian Kästner, and Christian Lengauer. “Language-independent and automated software composition: The FeatureHouse experience”. In: *IEEE Trans. Softw. Eng.* 39.1 (2013), pp. 63–79 (cit. p. 469).

[4] D Batory, J.N Sarvela, and A Rauschmayer. “Scaling step-wise refinement”. In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371 (cit. p. 469).

[5] B. Behringer, J. Palz, and T. Berger. “PEoPL: Projectional editing of product lines”. In: *International Conference on Software Engineering*. ICSE. 2017 (cit. pp. 466, 467, 472, 476, 480).

[6] Thorsten Berger, Stefan Stanciulescu, Ommund Ogaard, Oystein Haugen, Bo Larsen, and Andrzej Wąsowski. “To connect or not to connect: Experiences from modeling topological variability”. In: *SPLC*. 2014 (cit. p. 468).

[7] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. “Efficiency of projectional editing: A controlled experiment”. In: *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2016 (cit. pp. 471, 472, 476).

[8] Jean Bézivin, Frédéric Jouault, and David Touzet. “An introduction to the ATLAS model management architecture”. In: *Rapport de recherche* 5 (2005), pp. 10–49 (cit. p. 471).

[9] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. “A compositional framework for systematic modeling language reuse”. In: *MODELS*. 2020 (cit. pp. 465, 467, 471, 473, 480).

[10] Fabien Campagne. *The MPS Language Workbench Volume II: The Meta Programming System*. CreateSpace Independent Publishing Platform, 2016 (cit. p. 480).

[11] Magnus Christerson and Henk Kolk. *Domain Expert DSLs*. A talk at QCon London 2009. 2009. URL: http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk (cit. p. 479).

[12] Paul C. Clements and Charles Krueger. "Point/counterpoint: Being proactive pays off—eliminating the adoption". In: *IEEE Software* 19.4 (2002), pp. 28–30 (cit. p. 475).

[13] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge Into Tools*. CRC Press, 2016 (cit. pp. 479, 480).

[14] Krzysztof Czarnecki and Michał Antkiewicz. "Mapping features to models: A template approach based on superimposed variants". In: *GPCE*. 2005 (cit. p. 473).

[15] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000 (cit. p. 479).

[16] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. "EAST-ADL: An architecture description language". In: *IFIP World Computer Congress, TC 2*. Springer. 2004 (cit. p. 459).

[17] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *OOPSLA*. 2007 (cit. p. 480).

[18] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. "Language composition untangled". In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. 2012, pp. 1–8 (cit. p. 479).

[19] Sebastian Erdweg et al. "The state of the art in language workbenches". In: *SLE*. 2013 (cit. p. 472).

[20] Luca Favalli, Thomas Kühn, and Walter Cazzola. "Neverlang and FeatureIDE just married: Integrated language product line development environment". In: *24th ACM Conference on Systems and Software Product Lines*. 2020 (cit. p. 472).

[21] Simon Fürst et al. "AUTOSAR – a worldwide standard is on the road". In: *14th International VDI Congress Electronic Systems for Vehicles*. Vol. 62. 2009, p. 5 (cit. p. 459).

[22] Antonio Garmendia, Manuel Wimmer, Esther Guerra, Elena Gómez-Martínez, and Juan de Lara. "Automated variability injection for graphical modelling languages". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2020 (cit. pp. 461, 466, 469, 472).

[23] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. "Engineering tagging languages for DSLs". In: *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015 (cit. p. 469).

[24] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. "Analysing meta-model product lines". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2018. 2018 (cit. p. 473).

[25] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. "Property satisfiability analysis for product lines of modelling languages". In: *IEEE Transactions on Software Engineering* (2020) (cit. p. 473).

[26] Ines Hajri, Arda Goknil, Lionel C Briand, and Thierry Stephany. “Configuring use case models in product families”. In: *Software & Systems Modeling* 17.3 (2018), pp. 939–971 (cit. p. 459).

[27] Øystein Haugen and Birger Møller-Pedersen. “Configurations by UML”. In: *European Workshop on Software Architecture*. Springer. 2006, pp. 98–112 (cit. p. 468).

[28] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. “Adding standardized variability to domain specific languages”. In: *Proceedings of the 2008 12th International Software Product Line Conference*. SPLC ’08. 2008 (cit. pp. 468, 472, 473).

[29] Øystein Haugen and Ommund Øgård. “BVR – better variability results”. In: *International Conference on System Analysis and Modeling*. Springer. 2014, pp. 1–15 (cit. pp. 468, 473).

[30] Oystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. “CVL: Common variability language”. In: *17th International Software Product Line Conference*. SPLC. 2013 (cit. pp. 468, 473).

[31] Görel Hedin. “An introductory tutorial on JastAdd attribute grammars”. In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Springer, 2011, pp. 166–200 (cit. p. 480).

[32] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58 (cit. pp. 471, 480).

[33] Florian Heidenreich, Jan Kopcsek, and Christian Wende. “FeatureMapper: Mapping features to models”. In: *Companion of the 30th International Conference on Software Engineering*. 2008, pp. 943–944 (cit. pp. 468, 473).

[34] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. “Software language engineering in the large: Towards composing and deriving languages”. In: *Computer Languages, Systems & Structures* 54 (2018), pp. 386–405 (cit. pp. 465, 472, 479).

[35] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. “Empirical assessment of MDE in industry”. In: *ICSE*. 2011 (cit. p. 479).

[36] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 1990 (cit. p. 459).

[37] Christian Kästner and Sven Apel. “Integrating compositional and annotative approaches for product line engineering”. In: *McGPLE*. 2008 (cit. pp. 467, 475).

[38] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. “DeltaJ 1.5: Delta-oriented programming for Java 1.5”. In: *PPPJ*. 2014 (cit. p. 469).

[39] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: A framework for compositional development of domain specific languages”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.5 (2010), pp. 353–372 (cit. pp. 471, 480).

[40] Jacob Krueger and Thorsten Berger. “An empirical analysis of the costs of clone- and platform-oriented software reuse”. In: *28th ACM SIGSOFT*

International Symposium on the Foundations of Software Engineering (FSE). 2020 (cit. p. 475).

[41] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. "Choosy and picky: Configuration of language product lines". In: *Proceedings of the 19th International Conference on Software Product Line*. 2015 (cit. p. 473).

[42] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. "Introducing variability into aspect-oriented modeling approaches". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2007, pp. 498–513 (cit. p. 471).

[43] Michael Lienhardt, Ferruccio Damiani, Lorenzo Testa, and Gianluca Turin. "On checking delta-oriented product lines of statecharts". In: *Science of Computer Programming* 166 (2018), pp. 3–34 (cit. p. 459).

[44] Max Lillack, Stefan Stanciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. "Intention-based integration of software variants". In: *41st International Conference on Software Engineering*. ICSE. 2019 (cit. p. 476).

[45] Lukas Linsbauer, Felix Schwaegerl, Thorsten Berger, and Paul Gruenbacher. "Concepts of variation control systems". In: *Journal of Systems and Software* 171 (2021), p. 110796 (cit. p. 468).

[46] Wardah Mahmood, Daniel Strueber, Anthony Anjorin, and Thorsten Berger. "Effects of variability in models: A family of experiments". In: *Empirical Software Engineering* (2022) (cit. p. 475).

[47] Raul Medina-Mora and Peter H. Feiler. "An incremental programming environment". In: *IEEE Trans. Softw. Eng.* 7.5 (Sept. 1981), pp. 472–482 (cit. p. 479).

[48] David Méndez-Acuña, José A Galindo, Benoit Combemale, Arnaud Blouin, and Benoit Baudry. "Reverse engineering language product lines from existing DSL variants". In: *Journal of Systems and Software* 133 (2017), pp. 145–158 (cit. p. 480).

[49] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. "LISA: An interactive environment for programming language development". In: *International Conference on Compiler Construction*. Springer. 2002 (cit. p. 471).

[50] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. "Weaving variability into domain metamodels". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, pp. 690–705 (cit. p. 470).

[51] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. "Multi-view editing of software product lines with PEoPL". In: *40th International Conference on Software Engineering (ICSE), Demonstrations Track*. 2018 (cit. pp. 466, 480).

[52] Radu Muschevici, José Proença, and Dave Clarke. "Feature nets: Behavioural modelling of software product lines". In: *Software & Systems Modeling* 15.4 (2016), pp. 1181–1206 (cit. p. 459).

[53] David Notkin. "The GANDALF project". In: *J. Syst. Softw.* 5.2 (May 1985) (cit. p. 479).

[54] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005 (cit. p. 475).

[55] Thomas W. Reps and Tim Teitelbaum. “The synthesizer generator”. In: *Proc. SDE*. 1984 (cit. p. 479).

[56] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. “Lifting model transformations to product lines”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014 (cit. p. 469).

[57] Pablo Sánchez, Neil Loughran, Lidia Fuentes, and Alessandro Garcia. “Engineering languages for specifying product-derivation processes in software product lines”. In: *International Conference on Software Language Engineering*. Springer. 2008, pp. 188–207 (cit. p. 469).

[58] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. “Delta-oriented programming of software product lines”. In: *SPLC*. 2010 (cit. p. 469).

[59] Michael Schulze and Stefan Kuntz. “AUTOSAR and variability”. In: *ATZextra worldwide* 18.9 (2013), pp. 108–110 (cit. p. 459).

[60] Felix Schwägerl and Bernhard Westfechtel. “Integrated revision and variation control for evolving model-driven software product lines”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3373–3420 (cit. p. 468).

[61] Bran Selić. “The pragmatics of model-driven development”. In: *IEEE Software* 20.5 (2003), pp. 19–25 (cit. p. 479).

[62] Charles Simonyi. “The death of computer languages, the birth of intentional programming”. In: *Proc. NATO Science Committee Conference*. 1995 (cit. p. 479).

[63] Charles Simonyi, Magnus Christerson, and Shane Clifford. “Intentional software”. In: *Proceedings of OOPSLA*. 2006 (cit. p. 479).

[64] Daniel Strueber, Anthony Anjorin, and Thorsten Berger. “Variability representations in class models: An empirical assessment”. In: *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2020 (cit. p. 475).

[65] Jacques Thomas, Christian Dziobek, and Bernd Hedenetz. “Variability management in the AUTOSAR-based development of applications for in-vehicle systems”. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. 2011, pp. 137–140 (cit. p. 459).

[66] Juha-Pekka Tolvanen and Steven Kelly. “How domain-specific modeling languages address variability in product line development: Investigation of 23 cases”. In: *SPLC*. Paris, France, 2019, 24:1–24:9 (cit. p. 475).

[67] Edoardo Vacchi and Walter Cazzola. “Neverlang: A framework for feature-oriented language development”. In: *Computer Languages, Systems & Structures* 43 (2015), pp. 1–40 (cit. pp. 471, 480).

[68] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. “Variability support in domain-specific language development”. In: *International Conference on Software Language Engineering*. Springer. 2013, pp. 76–95 (cit. p. 471).

[69] Mark G.J. Van den Brand, Jeroen Scheerder, Jurgen J Vinju, and Eelco Visser. “Disambiguation filters for scannerless generalized LR parsers”. In: *International Conference on Compiler Construction*. 2002 (cit. p. 472).

[70] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007 (cit. p. 475).

[71] Anatoly Vasilevskiy, Øystein Haugen, Franck Chauvel, Martin Fagereng Johansen, and Daisuke Shimbara. “The BVR tool bundle to support product line engineering”. In: *SPLC*. 2015 (cit. pp. 468, 473).

[72] Markus Völter. *DSL Engineering. Designing, Implementing and Using Domain Specific Languages*. 2013. URL: http://www.dslbook.org (cit. p. 480).

[73] Markus Völter. “Language and IDE modularization and composition with MPS”. In: *GTTSE*. LNCS. Springer, 2011 (cit. p. 480).

[74] Markus Völter and Sascha Lisson. “Supporting diverse notations in MPS’ projectional editor”. In: (2014) (cit. p. 480).

[75] Markus Völter and Vaclav Pech. “Language modularity with the MPS language workbench”. In: *34th International Conference on Software Engineering (ICSE)*. 2012 (cit. pp. 471, 472).

[76] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. “Towards user-friendly projectional editors”. In: *International Conference on Software Language Engineering*. Springer. 2014, pp. 41–61 (cit. pp. 472, 476, 479).

[77] Markus Völter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. “Efficient development of consistent projectional editors using Grammar Cells”. In: *9th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. 2016 (cit. p. 476).

[78] Markus Völter and Eelco Visser. “Language extension and composition with language workbenches”. In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA. 2010 (cit. pp. 470, 472, 479).

[79] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. “Improving domain-specific language reuse with software product line techniques”. In: *IEEE Software* 26.4 (2009), pp. 47–53 (cit. p. 473).

[80] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. “Towards a UML profile for software product lines”. In: *Software Product-Family Engineering*. 2004 (cit. p. 472).

[81] Tewfik Ziadi and Jean-Marc Jézéquel. “Software product line engineering with the UML: Deriving products”. In: *Software Product Lines*. Springer, 2006, pp. 557–588 (cit. p. 472).

[82] Steffen Zschaler et al. “VML* – a family of languages for variability management in software product lines”. In: *International Conference on Software Language Engineering*. Springer. 2009 (cit. pp. 469, 472).