Domain-Specific Languages

Effective Modeling, Automation, and Reuse

With examples in Scala, ATL, Alloy, C#, F#, Groovy, Java, JavaScript, Kotlin, OCL, Python, QVT, and Xtend

Including 185 exercises for effective teaching & learning

Andrzej Wąsowski and **Thorsten Berger**

13/01/2021, v. 0.11.0 (work-in-progress)

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

Contents

Preface

This book has been jointly developed in a series of project-based courses on Model-Driven Software Engineering (MDSE) and Domain-Specific Languages (DSLs) at the IT University of Copenhagen in Denmark, at Chalmers University of Technology, and the University of Gothenburg in Sweden.

Acknowledgements

We thank Rolf-Helge Pfeiffer, Stefan Catalin Stanciulescu, Alexandru Florin Iosif-Lazar, and Kasper Hansen for suggestions of improvement and for corrections. Alexandru Florin Iosif-Lazar has written an early version of the configurator project specification. Kasper Hansen has written the original solution to the programming language primer exercise. We should also credit the students who have suffered the courses (in excess of 300 now) for being the main reason to create the material and for being the main testers of it.

We thank Marek Furák, Francisco Martínez Lasaca, Abdulrashid Mas'ab Mohammed, Christoffer Stougaard Pedersen, Robin Bellini Olsson, and Cem Turan for contributing corrections. Special thanks go the the participants of the respective course at Chalmers: Oscar Jönsson, Robert Palm, Jonatan Gustafsson, for spending time with us reporting feedback and suggestions for improvement.

Anders Fischer Nielsen supported some .NET development. Karol Wąsowski implemented the first Kotlin example. Georg Hinkel has supported us with using NFM.

Twitter discussions participants that led to identifying details in the book: Jean Bezivin, Jeremy Gibbons, Vadim Zaytsev, Jurgen Vinju, Tijs van der Storm, Erik Meijer, Miëtek Bak, typeswitch

Everything is a model.
(Bézivin, 2005a)

1 Using Modeling Languages

1.1 Why Modeling?

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 in these areas. Engineers build them to assess system properties early or to steer construction, production, and servicing processes. For one system, usually different kinds of models have to be built, each of which 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 carry the main characteristic aspects of the domain. For example, telephone-network switching models are very 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: complexity of the problem, complexity of the solution design, and complexity of the system implementation or production. The understanding of complex problems, solutions, designs, and processes is possible thanks to *abstraction* (Selic, 2003). Abstraction is a 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 of the model. We can say that *"all models are wrong, but some are useful"* (Box and Draper, 1987). For instance, aesthetic information is typically not necessary to assess performance.

Models can not only *abstract* (or hide) information, but they can also *approximate* it. For instance, the Newtonian gravity model is sufficiently precise for applications in mechanical engineering. It is also widely applied, although we know that it is not precise. It is crucial that both abstraction and

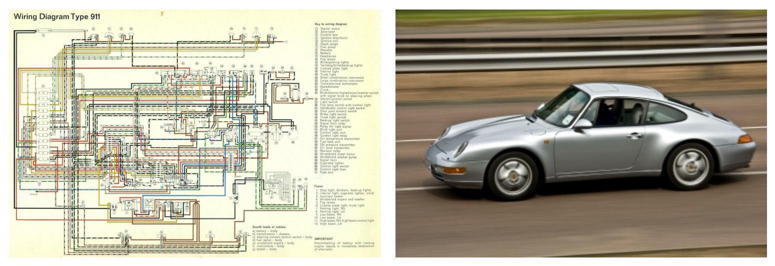


Figure 1.1: The electrical scheme of Porsche Carrera and the actual system. Note the abstraction of information

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.

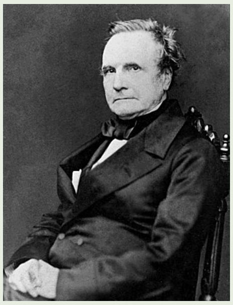
Models are increasingly electronic thanks to a rapid growth of computing technology. In fact, most of the engineering models are computer models today, even if they describe physical artifacts, such as buildings or diesel engines. Building computerized models is cheaper than building physical models. Specifically, it allows animation, simulation, and computation of non-structural and non-functional properties, such as weight and force of gravity on various elements.

Even though, computerized models have overtaken other engineering disciplines, models in software engineering are not as popular as elsewhere. Yet, the use of models in software engineering is 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 of using models in computing and of introducing purposeful domain-specific modeling languages (DSMLs, or DSLs for short) into the development of software systems.

There is no doubt that software engineers face the very same complexity problems that engineers in other disciplines have seen. In many ways, software systems are just as complex—commonly even more complex—than other achievements of engineering. Many commercial software systems have more lines of code than the “Jumbo Jet” 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.[1] In December 2014, the Linux kernel had about 15 million lines of code. This complexity is (partly) controlled using a configuration model and an automated build process (Berger et al., 2013). The Open Office productivity package had reached nine million lines of code in 2012.[2] Microsoft Windows is reported to have exceeded 50 million lines in 2003. *“Therefore, it seems obvious that*

[1] http://www.boeing.com/commercial/747family/pf/pf_facts.html, retrieved 2015/02/10
[2] http://www.openoffice.org/FAQs/build_faq.html, retrieved 2015/02/10

Model-Driven Engineering Prehistory



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 (Babbage, 1822).

Interestingly, Babbage's reasons to build the difference engine resembled the motivation of most automation projects, including model-driven software engineering. In the 19th century, mathematicians would calculate approximations of irrational functions manually. The results of these calculations (many-page long tables of numbers) would be then typeset by printers and printed on paper, so that engineers can use them in calculations.

Babbage considered this process to be tedious and error prone—an ideal candidate for automation. His machine would first calculate the values for the tables, but he did not stop there. He designed a printing back-end, so no errors would be introduced in typesetting. He even considered what paint colour should be used to minimize the number of errors. Today, when we search for opportunities to use MDSE, we also consider software development activities that are tedious and error prone, often associated with creating a lot of boilerplate code. We also want to use models and automation not only to derive early designs, but all the way to build effectively functioning systems end-to-end.

The British government showed interest in Babbage's project, believing that this could bring down the cost of computing the numeric tables. This is also the same argument that modern project managers use 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. 1.3.

Incidentally, the difference engine has been built twice in modern times, following the blueprints made by Babbage. One can appreciate it in the Science Museum in London and in the Computer History Museum in Mountain View, California.

software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques" (Selic, 2003), in order to combat the complexity.

1.2 Model-Driven Software Engineering

Software development is particularly suitable for the use of models. When building a car, there is a notable abstraction gap between a model and a real physical construction (recall Fig. 1.1). In computing, both models and systems are virtual digital objects, so this gap is much smaller. In software engineering, everything is in fact a model (Bézivin, 2005b). It is possible to refine models into a system in a continuous and often automatic manner, with much less effort than when designing cars or buildings. A computer program—more precisely, its source code in a programming language—is also a virtual object. The code is just yet another model that abstracts many aspects of a physical computation, but perhaps preserves more details

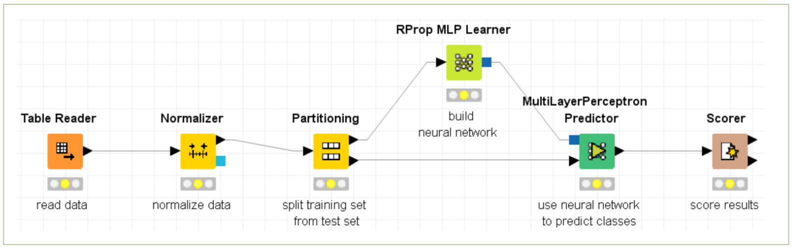


Figure 1.2: KNIME facilitates data analytics and AI development with a graphical DSL.

than other models (enough to run the computation). This proximity of models and programs allows to to make modeling the central paradigm in development—the main idea of Model-Driven Software Engineering.

Definition 1.2. Model-Driven Software Engineering (MDSE) *is a software-engineering methodology that focuses on creating and exploiting models to produce software. The focus is on models, modeling, and model analysis as opposed to programs and programming. MDSE relies heavily on automation to produce code, to analyze system properties, and to support development activities.*[3]

Organizations engineering software can adopt MDSE by either relying on off-the-shelf modeling techniques and languages that have been developed by others, or they can create their own languages and modeling infrastructure, thanks to powerful MDSE tools and techniques, which we will present in this book. As confirmed in surveys with industry (Hutchinson et al., 2011; Liebel et al., 2014; Whittle, Hutchinson, and Rouncefield, 2014), the most well-known off-the-shelf modeling language is the Unified Modeling Language (UML) (Petre, 2013; Fowler, 2004; Object Management Group, 2017), established and developed under the umbrella of the Object Management Group (OMG). UML boasts over 13 different languages (called diagram types), such as class diagrams or sequence diagram, which organizations can use without having to design languages and create language tooling, such as editors, to leverage MDSE. In many cases, however, it is desirable to create languages that are specifically tailored to an organization's need, such as a language that customers can use to configure a product, and so on. Let us now look into some off-the-shelf languages to get a better intuition into the looks and feels of such languages. Already in Chapter 2 we will start to design our first own language.

Example 1. KNIME[4] is an extensible platform for assembling data-analytics pipelines. It is used by data scientists for developing data analyses, visualizations, classifications, preprocessors, and many more tasks in data analytics—for instance, in pharmaceutical research, business intelligence, and financial decision making. At KNIME's core is a graphical DSL allowing non-

[3]Inspired by definitions of Selic (2003) and of Wikipedia editors (https://en.wikipedia.org/wiki/Model-driven_engineering, retrieved 2017/01/06).

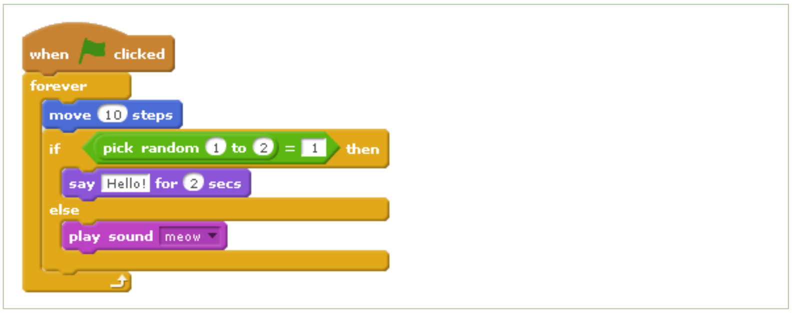


Figure 1.3: Scratch facilitates end-user development with a graphical DSL.

programmers to assemble the pipeline. Figure 1.2 depicts an example model, where a multi-layer perceptron (MLP) and a predictor for classifying image data is modeled. The figure nicely shows a typical pipeline, the nodes represent data-processing functionality, and the edges the data-flow. Specifically, the first node "Table Reader" obtains a dataset, which is then normalized in the second node "Normalizer," then partitioned into training set and test set (in our example, into 1/3 and 2/3, specified in the node properties that are not shown in the Figure), then fed into an "MLP Learner." Thereafter, a "Predictor" uses the learned neural network to do the actual classification of input (image) data on the test data, which is then compared with the original, ground-truth data in the node "Scorer," which creates, for instance, a confusion matrix and various statistics (e.g., recall, precision, F-measure). For an introduction in data analytics and, specifically, machine learning, we recommend the book of Rogers and Girolami (2016). In general, what this example illustrates is that given a proper language, users can focus on the actual domain (data analytics) without having to know the details of programming, such as knowing the exact APIs, writing glue code, and scripting the pipeline that pushes the data through the individual parts of the pipeline.

Example 2. Scratch[5] is another end-user oriented modeling language. Unlike KNIME, Scratch is an imperative language, composed of control blocks akin to programming languages. Its semantics (not the syntax) is reminiscent of control-flow graphs. The syntax is designed to meet the expectations of the target user group, primary-school children. Scratch programs resemble jigsaw puzzles (see Fig. 1.3). Using a domain-specific syntax that matches the user expectations and the problem at hand well is one of the key success factors in designing DSMLs. 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, since the user is not supposed to write general programs in it. It focused on game-like sprite programs that children can write.

[4] https://www.knime.com
[5] http://scratch.mit.edu

```
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*

Example 3. Google Protocol Buffers is 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 can be classified as a kind of 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 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 reordering and renaming the fields without changing the message format. The format also allows specifying optional elements with 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 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 would be reimplemented repetitively 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 the 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.

Using MDSE, we can express software design using concepts that are closer to the problem domain than to the implementation technology (Selic, 2003). We can stop talking about classes and loops, and instead consider business entities, cash-flow processes, and customers. In extreme cases, software can be tailored by domain experts or more technical users, when the modeling language used is designed with end-users in mind. A good example here are customizable enterprise systems, which are implemented by software engineers who are highly skilled in software architectures and programming, but customized by business-domain consultants who know more about enterprise architectures and business processes. Also many computer games allow end-user extensions through various "mod" packages implemented as domain-specific programs (models). However, for most systems the benefit is reachable more easily by raising the abstraction level at which the developers work—like in the Google Protocol Buffers and Rails examples.

Especially observe that in these two examples the models are mixed with code. Using modeling in software development does not exclude programming. Modeling is simply a more efficient way to program aspects of the systems that are well understood. For this reason, in practice, MDSE is often introduced into systems and domains that are already mature.

A common misunderstanding is that abstract models cannot be used effectively in software production, as they contain little information, not enough to generate systems from them. In all the above examples, automatic infrastructures complete the abstract models with concrete information, effectively turning them into programs. Once we add enough information to models, they effectively become programs, loosing all the advantages of modeling. This argument misses the fact that in MDSE there are two sources of information—the models are just one of them. The other one is the language definition (and then also the language implementation). A good modeling language captures the commonality of the domain in the language semantics and implementation, and let the aspects that vary across uses of the framework to be specified in models. KNIME, Scratch, and Protocol Buffers are extremely simple languages, yet one can derive

Terminology of MDSE

The terminology established in the field of **MDSE** is relatively diverse. Beginners often struggle with the many synonyms that are used for one concept, and also the different usages of terms. In this book, we will use the terms **MDSE** (Model-Driven Software Engineering) and **DSL** (Domain-Specific Language).

MDSE is an umbrella term for the whole field of using models to *engineer* software. This *engineering* does not only comprise the development of software assets—as referred to with **MDSD** (Model-Driven Software Development), but also other activities in the lifecycle of a software, such as evolution and quality assurance. **MDE** (Model-Driven Engineering) and **MDD** (Model-Driven Development) in principle correspond to **MDSE** and **MDSD**, but are supersets. They are broader approaches, not restricted to software, and comprise the engineering or development of further assets, such as hardware. **MDA** (Model-Driven Architecture) is often used synonymously to all these MDSE-related terms. However, it refers to a specific standard established by the Object Management Group (Object Management Group, 2014) describing a software-design process starting with domain modeling in order to obtain platform-specific models that can ultimately be executed (Frankel, 2003; Mellor, 2004).

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. **DSL**s should be understandable by a domain expert; their strength is the reduced expressiveness compared to **GPL**s. **DSML** (Domain-Specific Modeling Language) refers to a subset of **DSL**—languages used specifically for domain-specific modeling. Further subsets of **DSL**s are domain-specific markup languages (e.g., XHTML, MathML) and domain-specific programming languages (e.g., Perl, shell-script languages). Although the majority of examples in this book concerns **DSML**s—since we use and develop languages to *model* software—we will use the term **DSL**, since we also consider markup languages and will implement so-called internal **DSL**s, which are embedded into a host **GPL**.

Finally, terms recently gaining popularity are **Low-Code Platform** 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 (Richardson and Rymer, 2016; Hendriks, 2017).

complex systems out of their models. This is not unlike their programming languages, which also abstract details of computation that are common to all programs, in the compiler, in the execution platform, and in the hardware.

1.3 Model-Driven Software Engineering in Industry

The last decades have seen an increasing interest in modeling and MDSE among industrial practitioners and researchers. As such, we can rely on a large body of knowledge reporting about industrial adoption and practices related to MDSE. These publications typically strive to explain the domains and the circumstances in which MDSE is (successfully) applied, the usage and role of models, as well as the perceived benefits and challenges—or risks—encountered. It is probably a good idea if you read into some of these experience reports, to get a better understanding on industrial software engineering using MDSE with DSLs.

The most notable works reporting on industrial adoption and practices are probably the empirical studies of Hutchinson et al. (2011), Bone and Cloutier (2010), Liebel et al. (2014), Forward and Lethbridge (2008), and Torchiano et al. (2011), who survey or interview industrial practitioners about their use of MDSE. For instance, Forward and Lethbridge (2008) conduct a survey of 113 practitioners on attitudes towards modeling compared to plain, code-oriented development. Hutchinson et al. (2011) attempt to understand what impacts the benefits and attitudes towards MDSE by surveying 250 individuals in diverse organizations to respond to a survey, and interviewed in depth 22 professionals using MDSE in 17 different companies. Such empirical studies are complemented by literature reviews, such as Mohagheghi and Dehlen (2008) who identified 25 papers published between 2002 and 2007 that report on industrial experiences. There are also publications or presentations by well-known researchers who summarize their experiences in working with industry. For instance, Selić (2017) in a keynote talk from 2017 explains that over 70 papers exist that report on experiences with MDSE. But, already in 1997 Deursen and Klint (1997), based on industrial practice, discuss the role, benefits, risks, and how to mitigate risks of MDSE in practice. Finally, for obtaining more qualitative details, a large number of case studies and individual industrial experiences is available, many of which we will refer to in the remainder of this section.

According to just one survey (Hutchinson et al., 2011), 83 % of the 250 respondents think that MDSE is beneficial, while only 5 % disagree. In the following, we take a look at where MDSE is reportedly adopted, how models are used, and what the perceived or confirmed benefits and challenges are—referring to the relevant empirical studies and experience reports.

Where is MDSE Used? MDSE is used especially in domains where complex business logic co-exists with a lot of technical details that benefit from abstraction, where software should be reused, or where software needs to interact with hardware, whose characteristics (e.g., dependencies or behavior) need to be modeled. Primarily, as shown by Bone and Cloutier (2010) who surveyed N=128 practitioners, large and long-living software projects are more prone to adopting MDSE than short-living ones, where the additional effort would not always pay off. Interestingly, a survey by Torchiano et al. (2011) (N=155) shows that the use of modeling correlates positively with the company size: larger organizations model more. Nevertheless, there are reports that small companies also benefit from adopting MDSE, too (Cuadrado, Cánovas Izquierdo, and Molina, 2014).

The literature review of Mohagheghi and Dehlen (2008) provides documented adoption of MDSE in the domains *telecommunication* (Weigert and Weil, 2006; Staron, 2006; Baker, Loh, and Weil, 2005), *business applications and financial organizations* (Deng et al., 2003), *web applications* (Brambilla, Ceri, et al., 2005), *aerospace and defense* (Jouenne and Normand, 2005), as well as *embedded* (Trask et al., 2006) *and safety-critical systems* (Safa, 2006). The survey of Bone and Cloutier (2010) adds *automotive* (Broy, 2006), *IT*, *medical* (Mashariki, Bronner, and Kazanzides, 2007), and *space systems* (Eisenmann, Miro, and Koning, 2009). Other substantial experience reports worth mentioning concern the domains *railway technology* (MacDonald, Russell, and Atchison, 2005) and *eGovernment* (Mellegård et al., 2016). Especially the latter, as well as a report about Motorola Baker, Loh, and Weil (2005), provide rich and longitudinal data of companies' use of MDSE. Selić (2017) reports similar domains, adding *industrial automation* (Staron, 2006) and *office automation systems* (Trčka et al., 2011). Selic also lists large companies that have reportedly 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[6], Volkswagen, ABB, Deere & Co., FMC, Siemens, Alcatel-Lucent, Ericsson, Motorola, Nortel, Siemens, UBS, and SAP.

In addition to all these industrial experience reports, we can find many publications 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 demand, and likely also actual usage in (industrial) practice.

Robotics is generally seen as a field that highly benefits from MDSE technologies, especially from models that abstract over the hardware and many low-level movement control algorithms. Robotics software is also still mostly developed in an ad hoc way (Garcia, Strueber, Brugali, Berger, et al., 2020; Garcia, Pelliccione, et al., 2018), less systematic, and the respective control software is often hardly reusable (Garcia, Strueber, Brugali, Fava, et al., 2019). A notable survey on DSLs for robotics by Nordmann et al. (2016) identified 41 publications that present a robotics DSL. Their reference example is a kinematics DSL (Frigerio, Buchli, and Caldwell, 2011) developed to control robotic soccer players in a specific discipline within the RoboCup competition.[7] In fact, more robotics soccer DSLs exist, such as CABSL (Röfer, 2018), which can be used to program the behavior of specific soccer players (e.g., the goal keeper). Another example are DSLs for controlling humanoid robots, such as DANCE, presented by Huang and Hudak (2003). The notation is textual, but it was inspired by a "domain-specific notation" called Labanotation invented in 1928 (sic!) by a German dance artist and choreograph. 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, various DSLs have been presented to ease utilizing this technology. The survey

[6]In fact, Volvo Cars recruiters are known to ask for applicants' performance in the MDSE courses.

[7]http://www.robocup.org

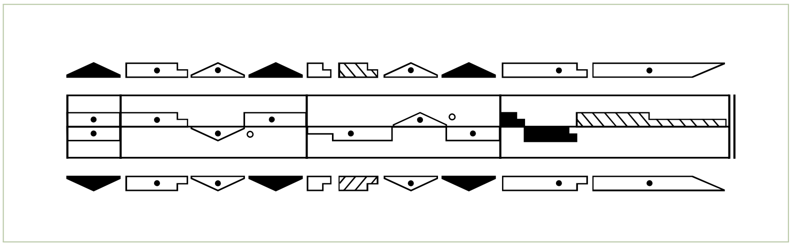


Figure 1.6: A dance choreography expressed in the visual DSL Labanotation (CC BY-SA 4.0, from Wikimedia Commons, author: Inigolv)

by Portugal, Alencar, and Cowan (2016) identifies seven DSLs, three of which are 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 (2000) gives an annotated overview of practice, technology, and motivation for using domain specific languages. Already in the year 2000, they listed over thirty DSLs documented in the literature, many in wide-spread 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 (2014) online.[8]

How are models used? Almost every programmer is using models in some way or another, depending on what is exactly seen as a model (Bézivin, 2005b). In the context of MDSE, we use models mainly for automating various engineering activities, beyond using models just for the purpose of documentation. The literature survey by Mohagheghi and Dehlen (2008) lists code generation, simulation, testing, and automatic test generation as the main reported usages. If we want to more quantitatively understand for what purpose models are actually used in industrial practice, only the survey by Liebel et al. (2014) of 112 professional developers in the embedded systems domain, provides hard empirical data. The following usages are ordered by frequency as mentioned by their surveyed developers.

Code generation. Software or significant parts of it can be automatically generated from models. Most importantly, this allows generating an *easily available basic infrastructure*. In this book, we will experience it when designing and implementing DSLs using MDSE methods. We will generate implementations of models, (de)serializers of models, instance generators for languages, tests, and editors for models, and well-formedness checkers for models. This means that working with models is more effective than working with code capturing similar information. The support mechanisms of a programming language know little about the information you are editing. For instance, serializing complex data structures using programming-language libraries usually does not work well; either it is not portable across program runs, or across machines, or introduces unacceptable changes

[8] https://github.com/slebok/yabib

to the data, or includes unnecessary runtime information. Similarly, a Java or C# editor can provide relatively little feedback about errors in the descriptions of business entities.

Code generation can also be used for actually executing the models. It is quite common to execute models by generating code that exhibits the actual semantics that the model should have. However, let us also note that for executing models, in practice, *interpretation* is a very good (and very often better) alternative to code generation. Interpretation is often easier to implement and test, while maintaining the same benefits. In Chapter 2 we will already write an interpreter for a language that controls a mobile robot.

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 simulating the behavior of the system before it is actually constructed, and finding design mistakes early. However, software can be executed very quickly if automatic generation is used, so there is no acute need to simulate. Since software is virtual and does not require physical production processes, it is available as soon as it is designed. This is much different for systems mixing hardware and software, where simulation of hardware models allows execution of software before hardware is available. It also allows to explore environmental conditions that are hard to reproduce in practice (for instance unsafe situations).

Furthermore, 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 observing the system 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 implemented and to mock users behavior. Alternatively, specifications of system behavior (models) can be linked to a running system for runtime monitoring. Errors would be 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 becomes much easier.

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.

Documentation and information. Models provide excellent documentation, given their proximity to the domain. Often, they are self-documenting (Deursen, Klint, and Visser, 2000) and can be directly used as documentation, or embedded into such, regardless 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 by talking about code, non-technical rules are usually excluded from such, so using models can easy interactions. However, documentation and information, even though, reported as the third most frequent usage by Liebel et al. (2014), is never the prime usage in MDSE, since we strive to use models for automation.[9]

Model checking and verification. System models can be verified, for example, for safety properties. Since code generation is used, we are highly confident that properties of the model are also properties of the final system. Model checking and verification is predominantly used for established modeling languages (such as MATLAB Simulink), since developing model verification infrastructure is unfortunately quite expensive, and requires advanced expertise. For this reason, verification tools rarely exist for project specific languages. In the survey of Liebel et al. (2014), model checking and verification is elicited in a more fine-grained way, comprising (most frequent usage first): structural consistency checks, behavioral consistency checks, timing analysis, formal verification, safety compliance checks, and reliability analysis.

Test-case generation. If your models describe the possible behaviors of your system, then you can use them to generate test cases. Consider, for instance, finite state machine diagrams (one of our running examples, which we will introduce in Example 6 in Sect. 3.2). Such models describe the possible sequences of states that a system can execute. Given this information, you can generate test cases, especially the input data for the system, then run the system using this input data and observe that the output adheres to the information in the model. Of course, there are techniques to extract behavior from source code, but that is usually less accurate, cumbersome, and the code could actually be incorrect. Importantly, test cases derived from the code-under-test are much less likely to identify bugs (they cannot find functional bugs, for instance). An explicitly and independently created model is more trustworthy and therefore allows creating test cases that test what the domain experts had in mind.

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

[9]Liebel et al. (2014) scoped their survey to so-called mode-based engineering (MBE), which is a form of engineering that more loosely advocates the use of models. In MDSE instead, models *drive* the development.

Model-Based system integration. Finally, let us mention a use case that was not directly listed by Liebel et al. (2014), 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 models in 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 what MDSE is about and how models are used, let us discuss what an organization can gain by adopting MDSE. We order the benefits discussed in the literature by the frequency in which they were reported in the survey by Liebel et al. (2014) (N=112 professional developers in the embedded systems domain)—the only study known to us that systematically elicits many different benefits from practitioners.

Improved quality. The most frequently reported benefit according to Liebel et al. (2014) is quality improvement. First, generated code is typically of high quality. A substantial effort is put into the design of a code generator, and any fix of a mistake there, immediately improves the quality of all the generated code, reducing errors for all users. Second, simulation allows to catch and fix errors early, and potentially exercises more system behavior than could be done in later tests (e.g, by executing the software), essentially finding more errors. Third, models improve the quality of requirements in the sense that some requirements can be expressed within the model (Biffl, Mordinyi, and Schatten, 2007), which allows finding errors in requirements, checking completeness of requirements, or enhancing the clarity of requirements—as opposed to requirements solely expressed in natural language. Improved quality was also found in a controlled experiment by Kieburtz (2000), where the error reduction and productivity improvement (explained shortly) was statistically significant.

Mohagheghi and Dehlen (2008) emphasize two case studies about Motorola (Weigert and Weil, 2006; Baker, Loh, and Weil, 2005) and France Telecom. In addition to finding errors early, their survey finds that fewer code inspections were necessary in these cases. "For example, 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." (Baker, Loh, and Weil, 2005)

Improved reusability. Reuse of software means that you want to take a piece of software you developed and modify it to fit another purpose or context (Deursen, Klint, and Visser, 2000). For instance, you want to reuse a software developed for a specific hardware (e.g., a robot) for another hardware. The second most-frequently mentioned benefit according to

(Liebel et al., 2014) is the improved reusability of software. The previously mentioned Motorola case study also reports "reuse of designs and tests between platforms or releases" (Weigert and Weil, 2006) as a prime benefit. 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 (or instantiate a new one) and re-generate code or just run the interpreter. It is just easier to specify models than to modify code (Selic, 2003), as well as models foster knowledge conservation and reuse (Deursen, Klint, and Visser, 2000). Furthermore, since models typically abstract over hardware, one can write different generators or interpreters for different hardware platforms: more-driven software is more easily *retargetable*. Many modeling languages have been defined to specifically foster use, by allowing to describe the future modifications. They are often known as variability modeling languages. We will return to them in Chapter 8 when we talk about software product lines (Apel et al., 2013; Czarnecki and Eisenecker, 2000; Czarnecki, Bednasch, et al., 2002), which are portfolios of system variants in a specific domain.

Improved reliability. This third most frequently mentioned benefit (Liebel et al., 2014) is a consequence of automation (Deursen, Klint, and Visser, 2000; Selic, 2003) and reuse of expert knowledge for generating code. According to (Selic, 2003): "[...] 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 by Kieburtz et al. (1996). Reliability was also a prime benefit observed in the Motorola case study (Weigert and Weil, 2006), since: (i) insecure or unreliable coding practices can be avoided, (ii) specific, more 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 helps in assessing reliability.

Improved traceability. As stated above, models can be used for establishing and exploiting traceability. However, already by using MDSE, traceability is obtained as a kind of by-product (Winkler and Pilgrim, 2010). Especially when transforming models into other models, model transformation engines produce traces between the models automatically—in other words, they create and maintain a trace model, which can be queried. Furthermore, when many requirements can be expressed as part of the model, traceability is naturally improved. Finally, models foster the comprehension of change impacts when the system is changed (Deursen and Klint, 1997)—another traceability-related improvement.

Improved maintainability. While maintainability was only the fourth most frequently mentioned benefit by respondents of Liebel et al. (2014), many other works report maintainability and productivity increase as prime benefits of MDSE (Deursen, Klint, and Visser, 2000; Selic, 2003; Deursen and Klint, 1997; Kieburtz et al., 1996).

Various publications explain this benefit as follows (Forward and Lethbridge, 2008; Hutchinson et al., 2011; Deursen and Klint, 1997). First, software defined in domain terms is easier to maintain. Models are easier to understand than low-level code, and they can serve the role of documentation at times. It is easier to introduce new developers to tailor the systems using an abstract DSL instead of changing the low-level code, since most DSLs ensure that the changes stay within assumed design invariants. Collaboration and coordination among developers is improved through models (Forward and Lethbridge, 2008). MDSE allows easier modifications (Forward and Lethbridge, 2008) and comprehension of change impacts. Models and their languages more explicitly represent domain-specific knowledge, which is also represented in a platform-independent manner (Deursen and Klint, 1997). The latter also enhances *system portability* (Hutchinson et al., 2011)—another benefit of MDSE that has already been reported in 1988 by Herndon and Berzins, 1988.

Yet, it is actually interesting that the survey respondents (Liebel et al., 2014) are a bit less convinced about maintainability than the other benefits. An earlier survey of Forward and Lethbridge (2008) indicates some ambivalence: respondents found changeability better, but bug fixing was perceived as harder. We return to this when talking about risks of MDSE below.

Improved productivity. The reasons for better productivity are mostly the same as for improved maintainability: better understandability through abstraction and domain-orientation (various roles, including domain experts can understand the models) (Deursen, Klint, and Visser, 2000; Selic, 2003). Systems can be created much faster, and sometimes they can even be instantiated by non-technical domain experts who create the models and then initiate code generation and automated deployment (Selic, 2003). Such generated systems are also more usable and also likely to meet the original requirements (Forward and Lethbridge, 2008).

Huge productivity gains are quoted by practitioners of MDSE in interviews (Hutchinson et al., 2011): at least two-, but even eight-fold! The interviewees, however, explain that the increases are sometimes hidden from the management to protect against budget cut-downs (sic!).

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 visualizes examples of factors that influence various aspects of productivity (e.g., code development time), maintainability, and portability according to Hutchinson et al. (2011) (slightly revised and extended). The figure illustrates that

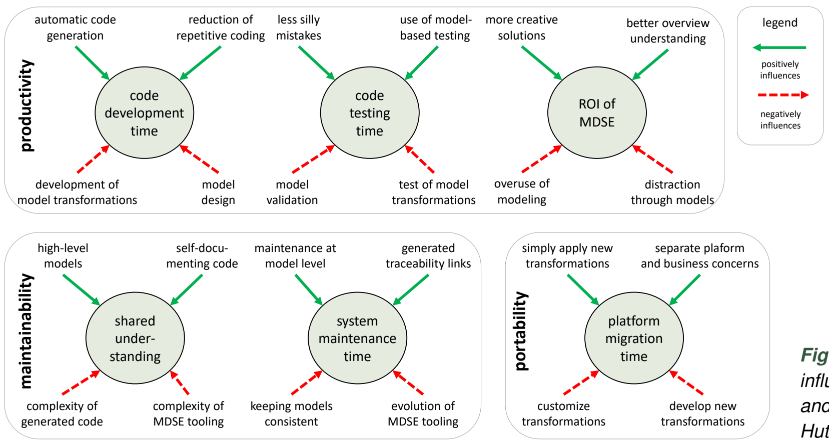

productivity
automatic code generation
reduction of repetitive coding
code development time
development of model transformations
model design
less silly mistakes
use of model-based testing
code testing time
model validation
test of model transformations
more creative solutions
better overview understanding
ROI of MDSE
overuse of modeling
distraction through models
legend
positively influences
negatively influences
maintainability
high-level models
self-documenting code
shared under-standing
complexity of generated code
complexity of MDSE tooling
maintenance at model level
generated traceability links
system maintenance time
keeping models consistent
evolution of MDSE tooling
portability
simply apply new transformations
separate plaform and business concerns
platform migration time
customize transformations
develop new transformations

Figure 1.7: *Illustrative influences of MDSE. Revised and extended from Hutchinson et al. (2011)*

positive and negative influences, which are also related to each other, should be taken into account when assessing MDSE. In practice, depending on the domain and project context, certain aspects will outweigh others.

Motivated by these influences, Hutchinson et al. (2011) study the impact of MDSE-related activities on productivity and maintainability. Even though, the eight activities they investigate are somewhat random, and it is not clear how representative they are, almost all activities except "use of models for testing" and model simulation/executable" models are clearly positively impacting maintainability and productivity. Furthermore, even though, as pointed out above, the majority of respondents considered using MDSE beneficial, a significant proportion (17–22 %) disagreed, which suggests some challenge (or risks) in adopting MDSE. The following are reported in the literature.

Return of investment. MDSE requires additional effort for engineering systems (including maintenance), and the risk is that this effort is too high and will not pay off when models are not useful enough (Torchiano et al., 2011; Deursen, Klint, and Visser, 2000). Another challenge can be the costs for education and training that are necessary for adopting MDSE (Liebel et al., 2014; Deursen, Klint, and Visser, 2000).

Half-baked adoption. There is also a risk when the potential of MDSE is not fully exploited, especially when models end up being solely used for documentation, not exploiting their full potential (Selic, 2003). Documentation too easily diverges from the reality.

Model and language quality. Selic (2003) sees low-quality models and non-adequate abstraction levels as significant risks. There should be enough emphasis on quality, since low-quality models can impact many different products. Deursen, Klint, and Visser (2000) also sees balancing challenges with models, for instance, balancing between generality and domain-specificity when developing DSLs. An important quality property for DSLs is also that they are properly scoped—so neither include too many nor too few concepts of the domain. A badly scoped DSL is a risk for MDSE (Deursen, Klint, and Visser, 2000). Finally, when code generation is used, the efficiency of the generated code is a considered risk (Deursen, Klint, and Visser, 2000), 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 is that inconsistencies arise when consistency is not continuously maintained (Forward and Lethbridge, 2008). To this end, the authors request better facilities for traceability and partial updates or co-evolution. Improved embedded modeling facilities (within code) could also alleviate this risk.

Tooling. Quality of tools is definitely a problem, even though the situation has improved significantly over the last decade. Hutchinson et al. (2011) 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. (2014) further emphasize tool interoperability and tool usability.

Social and Economic Challenges. According to Selic (2012), complex social and economic issues are the most important and difficult to solve challenges for MDSE: "based on long-term experience in industry, it is my opinion that these non-technical issues are the more critical ones to overcome."

Bug Fixing. Finally, that bug fixing in the context of MDSE can be more challenging is reported in the study of Forward and Lethbridge (2008). While changeability is better, bug fixing is perceived as more difficult in MDSE. This challenge is confirmed by our own experience, especially in courses where students are new to the subject. Experience helps, and many problems are primarily adoption problems or fighting with some idiosyncrasies of tools in the beginning.

The most important insight about bug fixing and all the other risks is, however, that if software would have been developed without MDSE, the effort and number of bugs would have been much higher. Consider alone all the editors and model serialization or deserialization infrastructures we will automatically generate. For developers it is almost impossible to achieve the high-quality editors that MDSE tools will generate, or at least it would be a much higher effort to engineer them from scratch. This will be a quite abstract statement for you now, but after having used MDSE, we are very sure that you will confirm this insight.

1.4 Scope and Structure of the Book

This book discusses methods and techniques for designing and implementing DSLs. We cover domain analysis, design of the syntax and semantics, as well as implementation of code generators and interpreters. Our primary goal is to learn how to design high-quality languages at relatively little cost, so that they can be used in much smaller projects, or with a much smaller

MDSE and Agile Software Engineering

- agile processes en vogue
- however, programming still takes a lot of time
- incrementality
- automation for continuous testing, integration, and delivery (Fitzgerald and Stol, 2014)
- see also RE vs. agile (Kasauli et al., 2017)

user base than the examples above. We focus primarily on discussing technology and implementation, as we deeply believe that automation is key to the successful use of models in software projects.[10]

Further Reading

The idea of DSLs is usually tracked to the seminal paper of Landin (1966), although the paper is concerned more with a family of related languages, where differences are introduced by (possibly significant amounts of) syntactic sugar, rather than with creating special purpose languages. Landin's languages differ syntactically, but share the same expressiveness. The suggestive title, and the fact that it argues for need of diverse language syntaxes for various needs, is usually the reason why this paper is considered as the first mention of DSLs.

One of the most referenced books on MDSE, that really helped establishing MDSE as a field and made it known to practitioners, is the book by Stahl and Völter (2005). It addresses a real need of the market, but also excels in presenting the UML-based approach to DSLs. Using UML and stereotyping remains probably the easiest way to create (especially graphical) languages, even today. In his newer book, Voelter (2013) focuses on using and developing domain-specific languages. This book is more comprehensive than ours, but we focus on presenting the material in style and structure that is suitable for use in a university course, without assuming an extensive training in compiler theory.

A somewhat more standard presentation of DSL design is given by Fowler and Parsons (2011), who thoroughly and excellently discuss the patterns and guidelines for implementing and using DSLs. However, they are less focused on obtaining the implementation with the modern tools at low cost, so their implementation of DSLs is not as much model-driven as presented in our book.

Brambilla, Cabot, and Wimmer (2012) present a very good overview of model-driven development architectures, processes, and benefits. In our opinion, their book is very suitable for experienced software developers who appreciate the software engineering issues solved by MDSE, and who are trained in (programming-) language design and implementation.

Combemale et al. (2016) have authored one of the more recent books on MDSE. The book is devoted to learning language design for MDSE, n 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.

[10] Note to reviewers: A reading guide will be inserted here, when the book is finalized.

References

Apel, Sven et al. (2013). *Feature-Oriented Software Product Lines*. Springer.

Babbage, Charles (1822). *Note on the application of machinery to the computation of astronomical and mathematical tables*.

Baker, Paul, Shiou Loh, and Frank Weil (2005). "Model-Driven engineering in a large industrial context–motorola case study". In: *International Conference on Model Driven Engineering Languages and Systems*.

Berger, Thorsten et al. (2013). "A Study of Variability Models and Languages in the Systems Software Domain". In: *IEEE Transactions on Software Engineering* 39.12, pp. 1611–1640.

Bézivin, Jean (2005a). "On the unification power of models". In: *Software and Systems Modeling* 4.2, pp. 171–188. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007/s10270-005-0079-0.

– (2005b). "On the unification power of models". In: *Software and System Modeling* 4.2, pp. 171–188.

Biffl, Stefan, Richard Mordinyi, and Alexander Schatten (2007). "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, p. 6.

Bone, Mary and Robert Cloutier (2010). "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*.

Box, George and Norman Draper (1987). *Empirical Model-Building and Response Surfaces*. Wiley.

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool.

Brambilla, Marco, Stefano Ceri, et al. (2005). "Model-driven Design of Service-enabled Web Applications". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, pp. 851–856. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066265. URL: http://doi.acm.org/10.1145/1066157.1066265.

Broy, Manfred (2006). "Challenges in Automotive Software Engineering". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, pp. 33–42. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292. URL: http://doi.acm.org/10.1145/1134285.1134292.

Combemale, Benoit et al. (2016). *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press.

Cuadrado, Jesús Sánchez, Javier Luis Cánovas Izquierdo, and Jesús García Molina (Sept. 2014). "Applying Model-driven Engineering in Small Software Enterprises". In: *Sci. Comput. Program.* 89.PB, pp. 176–198. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.04.007. URL: http://dx.doi.org/10.1016/j.scico.2013.04.007.

Czarnecki, Krzysztof, Thomas Bednasch, et al. (2002). "Generative Programming for Embedded Software: An Industrial Experience Report". In: *GPCE*. Ed. by Don S. Batory, Charles Consel, and Walid Taha. Vol. 2487. Lecture Notes in Computer Science. Springer, pp. 156–172. ISBN: 3-540-44284-7.

Czarnecki, Krzysztof and Ulrich Eisenecker (2000). *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley.

Deng, Gan et al. (2003). “Model driven development of inventory tracking system”. In: *Proceedings of the oopsla 2003 workshop on domain-specific modeling languages*.

Deursen, Arie van, Paul Klint, and Joost Visser (2000). “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35.6, pp. 26–36. DOI: 10.1145/352029.352035. URL: http://doi.acm.org/10.1145/352029.352035.

Deursen, Arie and Paul Klint (1997). *Little Languages: Little Maintenance?* Tech. rep. Amsterdam, The Netherlands.

Eisenmann, Harald, Juan Miro, and Hans Peter Koning (2009). “MBSE for European Space-Systems Development”. In: *INSIGHT* 12.4, pp. 47–53.

Fitzgerald, Brian and Klaas-Jan Stol (2014). “Continuous software engineering and beyond: trends and challenges”. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pp. 1–9.

Forward, Andrew and Timothy C. Lethbridge (2008). “Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals”. In: *Proceedings of the 2008 International Workshop on Models in Software Engineering*. MiSE ’08.

Fowler, Martin (2004). *Uml Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional.

Fowler, Martin and Rebecca Parsons (2011). *Domain-Specific Languages*. Addison-Wesley.

Frankel, David S (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons.

Frigerio, Marco, Jonas Buchli, and Darwin G Caldwell (2011). “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*.

Garcia, Sergio, Patrizio Pelliccione, et al. (2018). “An Architecture for Decentralized, Collaborative, and Autonomous Robots”. In: *International Conference on Software Architecture (ICSA)*.

Garcia, Sergio, Daniel Strueber, Davide Brugali, Thorsten Berger, et al. (2020). “Robotics Software Engineering: A Perspective from the Service Robotics Domain”. In: *28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*.

Garcia, Sergio, Daniel Strueber, Davide Brugali, Alessandro Di Fava, et al. (2019). “Variability Modeling of Service Robots: Experiences and Challenges”. In: *13th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*.

Hendriks, Dre (2017). “The selection process of model based platforms”. MA thesis. Radboud University Nijmegen.

Herndon Jr., R. M. and V. A. Berzins (June 1988). “The Realizable Benefits of a Language Prototyping Language”. In: *IEEE Trans. Softw. Eng.* 14.6, pp. 803–809. ISSN: 0098-5589. DOI: 10.1109/32.6159. URL: http://dx.doi.org/10.1109/32.6159.

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

Hutchinson, John et al. (2011). “Empirical assessment of MDE in industry”. In: *ICSE*. Ed. by Richard N. Taylor, Harald Gall, and Nenad Medvidovic. http://doi.acm.org/10.1145/1985793.1985858. ACM, pp. 471–480. ISBN: 978-1-4503-0445-0.

Jouenne, Eric and Véronique Normand (2005). “UML Modeling Languages and Applications”. In: ed. by Nuno Jardim Nunes et al. Berlin, Heidelberg: Springer-Verlag. Chap. Tailoring IEEE 1471 for MDE Support, pp. 163–174. ISBN: 3-540-25081-6. URL: http://dl.acm.org/citation.cfm?id=2206963.2206982.

Kasauli, R. et al. (2017). “Requirements Engineering Challenges in Large-Scale Agile System Development”. In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pp. 352–361.

Kieburtz, Richard B. (2000). *Defining and Implementing Closed, Domain-Specific Languages*.

Kieburtz, Richard B. et al. (1996). “A Software Engineering Experiment in Software Component Generation”. In: *ICSE*. IEEE Computer Society, pp. 542–552.

Lämmel, Ralf (2014). *Yet another annotated SLEBOK bibliography*. URL: https://github.com/slebok/yabib.

Landin, Peter J. (1966). “The next 700 programming languages”. In: *Communications of The ACM* 9.3, pp. 157–166.

Liebel, Grischa et al. (2014). “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)*.

MacDonald, Anthony, Danny Russell, and Brenton Atchison (2005). “Model-driven development within a legacy system: an industry experience report”. In: *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. IEEE, pp. 14–22.

Mashariki, Amen Ra, LeeRoy Bronner, and Peter Kazanzides (2007). “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.

Mellegård, Niklas et al. (2016). “Impact of introducing domain-specific modelling in software maintenance: an industrial case study”. In: *IEEE Transactions on Software Engineering* 42.3, pp. 245–260.

Mellor, Stephen J (2004). *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional.

Mohagheghi, Parastoo and Vegard Dehlen (2008). “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.

Nordmann, Arne et al. (2016). “A Survey on Domain-Specific Modeling and Languages in Robotics”. In: *Journal of Software Engineering in Robotics (JOSER)* 7.1, pp. 75–99.

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

– (2017). *Unified Modeling Language Specification 2.5.1*. https://www.omg.org/spec/UML.

Petre, Marian (2013). “UML in practice”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, pp. 722–731.

Portugal, Ivens, Paulo S. C. Alencar, and Donald D. Cowan (2016). “A Survey on Domain-Specific Languages for Machine Learning in Big Data”. In: *CoRR* abs/1602.07637. arXiv: 1602.07637. URL: http://arxiv.org/abs/1602.07637.

Richardson, Clay and John R Rymer (2016). “Vendor landscape: The fractured, fertile terrain of low-code application platforms”. In: *FORRESTER, Janeiro*. URL: https://informationsecurity.report/Resources/Whitepapers/0eb07c59-b01c-4399-9022-dfc297487060_Forrester%20Vendor%20Landscape%20The%20Fractured,%20Fertile%20Terrain.pdf.

Röfer, Thomas (2018). “RoboCup 2017: Robot World Cup XXI. Lecture Notes in Artificial Intelligence”. In: Springer. Chap. CABSL – C-based agent behavior specification language.

Rogers, Simon and Mark Girolami (2016). *A First Course in Machine Learning, Second Edition*. 2nd. Chapman & Hall/CRC.

Safa, Laurent (2006). “The practice of deploying DSM, report from a Japanese appliance maker trenches”. In: *Proceedings of the 6th OOPSLA Workshop on Domain Specific Modeling (DSM’06)*.

Selic, Bran (2003). “The Pragmatics of Model-Driven Development”. In: *IEEE Software* 20.5. http://csdl.computer.org/comp/mags/so/2003/05/s5019abs.htm, pp. 19–25.

– (Oct. 2012). “What will it take? A view on adoption of model-based methods in practice”. In: *Software & Systems Modeling* 11.4, pp. 513–526.

Selić, Bran (2017). *Model-Based Software Engineering in Industry: Revolution, Evolution, or Smoke?* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.347.4206&rep=rep1&type=pdf.

Stahl, Thomas and Markus Völter (2005). *Model-Driven Software Development*. Wiley.

Staron, Miroslaw (2006). “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.

Torchiano, Marco et al. (2011). “Preliminary Findings from a Survey on the MD State of the Practice”. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Trask, Bruce et al. (2006). “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*. OOPSLA ’06. Portland, Oregon, USA: ACM, pp. 846–853. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176733. URL: http://doi.acm.org/10.1145/1176617.1176733.

Trčka, Nikola et al. (2011). “Integrated model-driven design-space exploration for embedded systems”. In: *Embedded Computer Systems (SAMOS), 2011 International Conference on*. IEEE, pp. 339–346.

Voelter, Markus (2013). *DSL Engineering. Designing, implementing and using domain specific languages*. URL: http://www.dslbook.org/.

Weigert, Thomas and Frank Weil (2006). “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, 8–pp.

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

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

2 Building Modeling Languages

Our intention is to automate the development of software applications in a given domain, by using models to describe essential characteristics of an application and using code generation or interpretation to produce the application automatically. In order to make this approach work, we will need a language whose models are suitable to describe systems in this domain. Before creating models (i.e., language instances) and executing or otherwise using them, we need to design the notation in which these models are expressed. Such a language is called a DSL (a *domain-specific language*), as you recall from Chapter 1. The task of creating the language is performed by a *language designer*.

The remainder of this book is about creating DSLs, which incorporate domain-specific concepts. While many of the technologies presented in this book can also be used for realizing GPLs (general-purpose languages), we focus on the specifics of DSLs (e.g., their limited expressiveness) and side-step all the 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 (Aho et al., 2006; Mogensen, 2011; Sestoft, 2012).

In this chapter, we will first motivate the need for building DSLs and then discuss their different parts, which you as a language developer need to realize. We describe the various variants of these parts, their characteristics, and the respective technologies for realizing the parts. As such, this chapter serves as an overview of the topics covered in great detail later.

2.1 The Need for Domain-Specific Languages

General-purpose programming languages include language constructs that are useful for creating algorithms or structuring software systems. As such, they are not tailored towards a particular domain,[1] but contain general computation-related concepts, such as *loops* or *conditionals*. To tackle an increasing complexity of software, programming languages have continuously evolved and over time become more and more abstract.

Let us take a look at the history of concepts in programming languages. The very first programming languages—machine languages—only contained instructions that the machine's processor could execute directly, such

[1] On a philosophical stance, one could actually call programming languages as being specific to the domain of computation or algorithm development, but we avoid taking a position here.

as *counting*, *reading registers*, or doing some basic *input/output* operations. These instructions were stored as numbers in binary form, in the memory of computers. Any control flow (jumps) in these languages used concrete memory addresses (so also numbers) to indicate program locations.

In order to address the challenge of writing (slightly) more advanced programs, the assembly languages were designed (Booth and Britten, 1947). Also known as assemblers (a homonym with the name of compilers for the assembly languages), they contained concepts such as *jumps* and limited *arithmetic expressions*. Numeric instruction codes have been replaced by human readable mnemonic names and named labels for jumps had been added over time. The introduction of an assembly language had dramatically raised the level of abstraction in programs, hiding the complex low-level aspects of the machine language. 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 high-level programming languages. Fortran (Backus, Beeber, et al., 1957) was the first one. Algol 60 (Backus, Bauer, et al., 1963) followed shortly after. Both were imperative languages, introducing concepts such as *loops*, *conditionals* and *recursion*. LISP (McCarthy, 1960), introduced around the same time, was the first functional language based on Church's lambda calculus with the concept *function* (function value) as the main building block of a LISP program. Finally, COBOL-1961 (1961) introduced concepts well-suited for business programming, including *macros* and *hierarchical data structures*. All high-level languages had as a goal moving the computation away from low-level aspects of how a machine operates, and expressing it in abstract terms, not necessarily directly represented in the hardware.

The next well-recognized evolution leap in programming languages was the introduction of object-oriented programming. Simula 67 (Dahl and Nygaard, 1967) 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. 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. Right now, Scala is likely the language combining most of the object-oriented and functional features. We use Scala a lot in the remainder of this book (as many programming styles can be realized in Scala, this allows us to avoid switching languages all the time).

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 in the history of programming the expressiveness has been much less important than abstraction capabilities.

2.2 Domain-Specific Languages

Recently, language designers find it increasingly difficult to develop better general-purpose abstractions on top of the existing general-purpose programming languages. One practical way to further raise the abstraction level is to introduce domain-specific languages. The idea is to introduce domain-specific concepts directly to the language, as first-class components. Instead of trying to find a general-purpose abstraction, one can design abstractions that extract the essential aspects of the domain and hide the inessential ones. Such a language can no longer be used for any applications, but only for those in the problem domain. However, the abstraction level for the programs (models) in the problem domain raises. Hudak (1996) touts DSLs as the *ultimate abstraction*.

Good examples of successful languages that abstract away substantial amounts of details, yet put a lot of power into 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 mechanism. Both expose high-level domain-specific concepts, for instance, document elements (HTML) and data relationships (SQL). Let us explicitly define what DSLs are:

Definition 2.1. *A* domain-specific language (DSL) *is a computer programming or modeling language of limited expressiveness focused on a particular domain (or a particular aspect of a particular domain).*[2]

Today's engineers can use modern tools, so-called *language workbenches* (Erdweg et al., 2013), to create languages for much smaller domains than SQL and HTML. Language workbenches allow to build DSLs and their infrastructure cost-effectively. So, the next leap in the history of programming languages is in hands of software architects responsible for particular projects. Perhaps in your hands. For the first time, it is feasible to *create* compilers solely to increase quality and efficiency of individual software projects and products.

The key advantage of DSLs is also their key limitation—they can only be used in a specific domain, in a specific context, or for a specific activity. The elements within a DSL need to pertain to a domain to be coherent enough and understandable for the DSL users. Otherwise, it would be too difficult to use the language. In fact, limiting the language elements in 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 of its stakeholders, including a set of*

[2] After Fowler and Parsons (2011)

concepts and terminology understood by practitioners in that area, and including knowledge of how to build software systems (or parts of software systems) in that area.[3]

That a DSL is limited to a domain also means that one needs to design and implement a new language to benefit from it in a new project area. This is an inherent tension: a language used to lower the cost for a particular development problem incurs an additional development cost for language engineering. Two forces act in favor of reducing this cost: distribution of cost across multiple projects, and advances in language technology.

Distribution of cost across multiple projects. MDSE with DSLs is a cost-effective improvement to a development process and architecture *if* the language and its infrastructure can be reused across multiple projects in the same domain. For instance, a consultancy that customizes ERP systems will benefit from a DSL in which said customizations can be expressed fast and maintained efficiently. A DSL for coordinating several autonomous robots will benefit the developers if they can reuse the same language in several similar robotics projects. MDSE will be beneficial within a *single* project, if concise domain-specific models can replace complex boilerplate code in *multiple* locations in code, which is the case of use of embedded SQL or LINQ for database access. In these examples, the cost of language development is carried only once, while the benefits are reaped multiple times.

Advances in software tools. Advances in software tools make language design a relatively accessible skill. They also allow the language design to be performed in a time-efficient manner, without dominating the cost of the primary development tasks. Any organization deciding to develop a DSL needs access to language designers. Traditionally, this was not a skill easily available in software-development houses. Language designers and implementers were rare, and they found jobs in compiler companies. However, thanks to the emergence of well-integrated *language workbenches*, implementing DSLs became much easier: it can be done quickly. It no longer requires specialized compiler knowledge. It can nowadays be undertaken by most software developers who have completed a software-engineering or computer-science education (or completed reading this book).

A language workbench is a tool that facilitates rapid engineering of DSLs. Language workbenches follow an architecture known as the *meta-modeling hierarchy* or *bootstrapping*. We explain this idea throughout the book, starting with Sect. 3.9.

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

Today, language workbenches are a rather mature technology, that has existed already since 1980 (Erdweg et al., 2013). Xtext, a modern tool for development textual DSLs, is discussed in detail in Chapter 4, where we talk

[3] After Czarnecki and Eisenecker (2000)

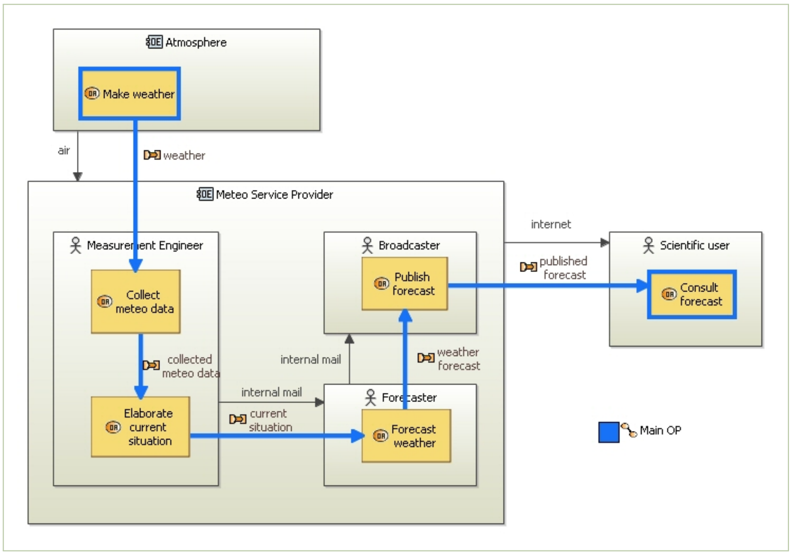


Figure 2.1: A service-definition model created in a graphical DSL designed in the language workbench Sirius (image from the Sirius gallery).

about designing concrete textual syntax. A popular tool for development of graphical DSLs is Sirius (Viyović, Mirjam Maksimović, and Perisić, 2014; Vujović, Mirjana Maksimović, and Perišić, 2014). The gallery of languages developed in Sirius[4] lists several interesting languages, including a service-definition notation, pertaining to the systems-engineering tool Capella, whose model is shown in Fig. 2.1. Even the standard graphical editor for Ecore models in Eclipse, discussed in Appendix B and used to create many figures in this book, is built using Sirius. We discuss Sirius in detail in **??**, devoted to languages with graphical syntax. We list the historical tools for both textual and visual languages in the Further Reading section, in the end of this chapter.

A third category of language workbenches, in addition to those for textual and for graphical languages above, are so-called *projectional language workbenches*. While they are conceptually similar to the workbenches for graphical languages, they support both textual and graphical syntax in an integrated way. The focus is slightly more on textual syntax than on the graphical ones, but using another technology, not a usual text editor, but a so-called projectional editor. As opposed to a regular text editor combined with a separate parser—the most common technology for realizing and using textual languages—a projectional editor does not need a parser. Instead, users directly work on a model or program. The abstract model 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 documents do). The user's editing operations directly change the nodes of the underlying representation, without any parsing involved. This technology allows for

[4] https://www.eclipse.org/sirius/gallery.html

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

We build domain-specific languages to write models and programs in these languages. So our main purpose is to work with instances of the languages. Interestingly, there is no single English noun that describes the different kinds of instances, and the word "instance" itself seems rather abstract and cryptic. The instances are typically called "models," but are often 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, as stated in Sect. 1.2, almost everything in software engineering is a model.

Kleppe (2009) 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), which is a first-class citizens in the context of MDSE. The actual name used for the instances is less important. Sadly, the word "mogram" has not caught on in the community. Voelter (2013) 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.

rendering of graphical notations and providing both textual and graphical notations within a single language, even within a single document. We discuss projectional languages and projectional editing in **??**.

2.3 What Is a Language Built Of?

Before we proceed to describe the overall process of building a language, let us consider what is it that we have to build; what is a language? Or more precisely: how language design and implementation can be split into smaller tasks? Indeed, most often we organize a language implementation in a chain of processors, in a so called pipeline architecture. The components are coarsely divided into two large groups, corresponding to the main parts of any language: the syntax and the semantics:

Definition 2.4. *The* Syntax *of a language is the definition of the principles and processes by which sentences are constructed in a particular language (Chomsky, 1957).*

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

Definition 2.5. *The* semantics *are the (study of) meanings of a language (Merriam-Webster).*

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

```
-> 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 `robot` language, describing a robot performing a random walk while avoiding obstacles

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 is loosely inspired by the architectural principle of reactive control, a specific way of controlling the behavior of robots.[a] 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 latter three modes are nested in the first mode (`RandomWalk`).

Each mode, besides other modes, can contain actions and reactions. *Actions* resemble regular programming language commands—they are immediately executed as the mode is activated, in the order from top to bottom. 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 (the two events in the example are: `obstacle` and `clap`) and then switches the mode to a new mode.

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 (`->`). The same 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 `->` symbol after the mode block indicates the successor mode (for instance, after `Avoid` the control moves to `MovingForward`). If a mode has a successor mode, then the control continues to the successor immediately after all actions have been executed. If a mode has no successor mode, the control stays in place, and awaits for any possible reaction triggers. Only reactions can now switch the active mode. In this simple language, it is impossible to

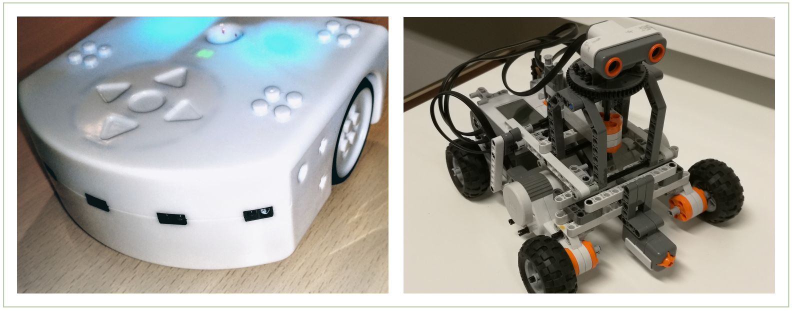


Figure 2.3: *Two example educational robots that are possible execution platforms for the robot control language used in Fig. 2.2: Thymio (left), Lego Mindstorms NXT (right)*

define multiple direct successor modes. 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 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.[5] The syntax is described with phrases of the following kind:

- Each mode, besides other modes, can contain 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. Semantics are defined over all models that are instances of a given language, but the specification of these semantics is only implemented once for the language, which determines the semantics of all models. For the example model in Fig. 2.2, the semantics is that of a random walk. Semantics also regulate detailed aspects of behavior, for instance, whether modes are pre-emptive or not. If you would specify in the semantics that modes are pre-emptive, this would mean that modes could be switched whenever a suitable reaction is triggered, leading to a new active mode, *even when* a computation was active in another mode. A non-pre-emptive semantics would not allow changing modes while actions are being executed. Statements like the following describe the semantics of our `robot` language:

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

The `robot` language is implemented in the book supplementary material in the project `mdsebook.robot`. 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?

[a] See also Chapter 14, especially Section 14.3, in the book of Matarić (2007)

[5] We will shortly distinguish between two kinds of syntax, the abstract and the concrete syntax, and you will observe that many different concrete syntax can exist for a language.

$$
\begin{array}{rcl}
\text{Mode} & \rightarrow & \texttt{'->'}?\ \ \text{Id}\ \ \texttt{'\{'}\ \Big(\ \text{Action} \mid \text{Reaction} \mid \text{Mode}\ \Big)^*\ \ \texttt{'\}'} \\
 & & \quad \Big(\ \texttt{'->'}\ \ \text{Id}\ \Big)? \\
\text{Reaction} & \rightarrow & \texttt{'on'}\ \ \text{Event}\ \ \texttt{'->'}\ \ \text{Id} \\
\text{Action} & \rightarrow & \Big(\ \text{AcDock} \mid \text{AcMove} \mid \text{AcTurn}\ \Big) \\
 & & \quad \Big(\ \texttt{'for'}\ \ \text{AExpr}\ \ \texttt{'s'} \mid \texttt{'at'}\ \ \texttt{'speed'}\ \ \text{AExpr}\ \Big)? \\
\text{AcDock} & \rightarrow & \texttt{'return'}\ \ \texttt{'to'}\ \ \texttt{'base'} \\
\text{AcTurn} & \rightarrow & \texttt{'turn'}\ \Big(\ \texttt{'right'} \mid \texttt{'left'}\ \Big)?\ \Big(\ \texttt{'by'}\ \ \text{AExpr}\ \Big)? \\
\text{AcMove} & \rightarrow & \texttt{'move'}\ \Big(\ \texttt{'forward'} \mid \texttt{'backward'}\ \Big) \\
\text{AExpr} & \rightarrow & \text{MinusMultExpr} \mid \text{PlusMultExpr} \\
\text{PlusMultExpr} & \rightarrow & \texttt{'+'}?\ \ \text{MultExpr}\ \Big(\ (\ \texttt{'+'} \mid \texttt{'-'}\ )\ \ \text{MultExpr}\ \Big)^* \\
\text{MinusMultExpr} & \rightarrow & \texttt{'-'}\ \ \text{MultExpr}\ \Big(\ (\ \texttt{'+'} \mid \texttt{'-'}\ )\ \ \text{MultExpr}\ \Big)^* \\
\text{MultExpr} & \rightarrow & \text{Atomic}\ \Big(\ (\ \texttt{'*'} \mid \texttt{'/'}\ )\ \ \text{Atomic}\ \Big)^* \\
\text{Atomic} & \rightarrow & \text{RndI} \mid \text{INT} \mid \texttt{'('}\ \ \text{AExpr}\ \ \texttt{')'} \\
\text{RndI} & \rightarrow & \texttt{'random'}\ \Big(\ \texttt{'('}\ \ \text{AExpr}\ \ \texttt{','}\ \ \text{AExpr}\ \ \texttt{')'}\ \Big)? \\
\text{Event} & \rightarrow & \texttt{'obstacle'} \mid \texttt{'clap'}
\end{array}
$$

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

In this book, we demonstrate the patterns, methods, and technologies for designing and implementing components of a language infrastructure. An entire 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 remaining parts of this chapter we demonstrate these key components using our example language. For each of the five aspects we discuss a definition, an example, a way to specify or implement it, and what tools are available to support it.

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.

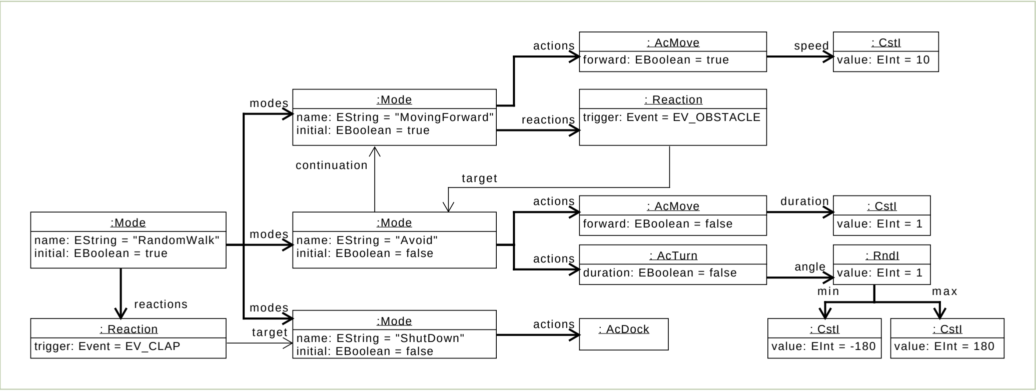


Figure 2.5: The abstract syntax of the random walk model. The bold lines indicate the tree structure

Specification. Any reader who has not studied language implementation surely appreciates how non-obvious it might be to build tools that work with concrete syntax. Automatically extracting structure and meaning from a flat sequence of characters in a file requires a non-trivial analysis. Fortunately, by now, this problem is extremely well understood, especially for simple languages like most DSLs are.

The concrete syntax for textual programming and modeling languages is typically defined using context-free grammars. An example of a concrete-syntax definition for the robot control language is shown 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, further followed by a list of actions, reactions, and modes enclosed in braces, and possibly followed by an identifier of the continuation mode. Such 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 more about grammars in Chapter 4.

Tools. Context-free grammars are interpreted by automatic tools, so-called *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 our robot control language it will, for instance, enforce that there has to be exactly one top-level mode (in which all the other modes shall be nested).

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 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. The reader will appreciate an approximate correspondence of nesting in this diagram with syntactic nesting in Fig. 2.2.

Specification. Since abstract syntax trees are data structures, they are defined using types, as all other data structures in programs. Presently, two ways of specification of abstract syntax are commonly accepted: class diagrams (so, class types in object-oriented programming 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 meta-modeling in Chapter 3.

Tools. In the implementation of DSLs, the abstract syntax is a *pivotal structure*: most language processing tools (parsers, importers, validators, converters, code generators, interpreters, visualizers, etc.) either produce or consume instances of 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.

Static semantics. The syntax of a language does define, which models in the language are correct. Still, just like for spoken languages, the syntactic correctness does not guarantee that a model makes sense. Consider the following English 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 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 our mobile robot control 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.

```
-> 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 }
}
```

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).*

Specification. Static semantic checks do depend much more on the idiosyncrasies of the designed language. Yet, typically, static semantics is specified by means of implementing a name analysis, type checking, and possibly a number of validity constraints. Definitions of static semantics are much less standardized than definitions of syntax, although many theories and frameworks exist. An example constraint for the mobile robot control language is shown in the bottom of Fig. 2.6. Much more complex and sophisticated static checks can be considered, although most language designers would limit themselves to properties that can be checked efficiently, to ensure a good usability of the language tools. We shall discuss definitions of static semantics extensively in Chapter 5.

Tools. Some aspects of static semantics can be handled by modern language workbenches. For instance, the Xtext framework has generic support for name analysis. If the static semantics is written in a specialized domain-specific language, then it can be automatically processed by tools for enforcing it. Examples of such tools include XSemantics (Bettini, 2013b), JetBrain's Meta-Programming System,[6] NaBL2[7] a part of the Spoofax workbench (Kats and Visser, 2010), PLT Redex in the Scheme community

[6] https://confluence.jetbrains.com/display/MPSD33/Typesystem
[7] http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/index.html

(Felleisen, Findler, and Flatt, 2009). The Object Constraint Language (OCL) (Object Management Group, 2010) provides a standardized formalism, with several available implementations for specifying first-order constraints, with transitive closure. It was largely created to give static semantics to languages (UML in the first place). Still, it is fairly popular to rely on manual implementations of the static semantics in GPLs, unlike for syntax parsing, where manual implementations have gone out of habit long ago. This is especially so, that proliferation of functional programming constructs in main stream programming languages, has made writing predicate concise and easy.

Dynamic semantics. Dynamic semantics defines the meaning of models in your language. This is typically done either by *interpreting* (e.g., executing, visualizing or calculating) the models, or by *translating* to other languages, for which the meaning is already known. Typically, the meaning can be defined in multiple ways for the same language. Indeed, we can talk of multiple meanings or interpretations for a single language. For our mobile robot control language, we can define the meaning by translating the models to a language executable on the target robot,[8] or by interpreting the models directly on a robot controller running a suitable robotics framework. One can also define the dynamic semantics abstractly, using some mathematical formalism. For the robot control language, a suitable meaning would be a set of execution traces (listing modes, actions and events). A possible execution 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 (infinite) set of all such execution traces would define the language formally (and very abstractly).

Figure 2.7 shows a small fragment of an interpreter for our language, implemented using Scala on top of the Robot Operating System (ROS).[9] We do not expect you to study how this is implemented, especially since large parts are omitted. We remark, though, the semantic gap between Figure 2.7 (robot program interpreter) and 2.2 (robot program). In the implementation of the interpreter, we notice 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 the ROS framework, however, they are not present in our small robot language. This illustrates the nature and need for domain-specific languages very clearly: a complex interpreter hides a large gap between the low-level language implementation and the input specification, 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.

[8] For example, Aseba script on a Thymio robot (https://www.thymio.org/en:asebalanguage-1-1), or URScript on a Universal Robot's arm (Universal Robots, 2015).

[9] See http://www.ros.org

```scala
class Interpreter(root: Mode) extends NodeMain {

  var lock: Lock = new ReentrantLock

  override def getDefaultNodeName(): GraphName =
    GraphName.of("mdsebook/robot/scala/interpreter")

  override def onStart(cn: ConnectedNode): Unit = {
    Thread sleep 1000
    var state = State(new mdsebook.robot.scala.Thymio(cn),
                  Map[Event, Reaction](), root)
    var listener = new MessageListener[LaserScan] {
      override def onNewMessage(msg: LaserScan): Unit =
        // obstacle event
        if (msg.getIntensities.sum > 0.09 && lock.tryLock)
          try state = state.processEvent (EV_OBSTACLE)
          finally lock.unlock
    }

    if (lock.tryLock) try {
      state.thymio.getProximityTopic addMessageListener listener
      state = state.activate
    } finally lock.unlock
  }
  // ...
}
```

Figure 2.7: *The core part of the interpreter for the mobile robot control language (dynamic semantics)*

The second, somewhat incidental, function of dynamic semantics is to further detect errors in models and programs, introducing any "last minute" validity checks at runtime—those that are necessary, but could not have been performed statically. The static semantics can only guarantee well-formedness of models and programs to a limited extent. It is well agreed between language experts that every non-trivial question about a program is undecidable.[10] For a programming language, a simple example of a property that is difficult to guarantee statically is the lack of divisions by zero, or whether a program throws an exception or not. For modeling languages, an undecidable property might, for instance, be whether there exists an instance of a class diagram satisfying all the diagram constraints and all the constraints that are part of the static semantics.

Incidentally, our mobile robot control language is so simple that, assuming the termination of all the actions, all interesting properties will be decidable—our models are always finite state. This is quite often the case for DSLs, and yet another reason to introduce and use DSLs—that is, the possibility to run more precise and effective semantic checks on the domain-specific models.

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

[10] This is formally grounded in the Rice's theorem, see Hopcroft, Motwani, and Ullman (2001) and Rice (1953).

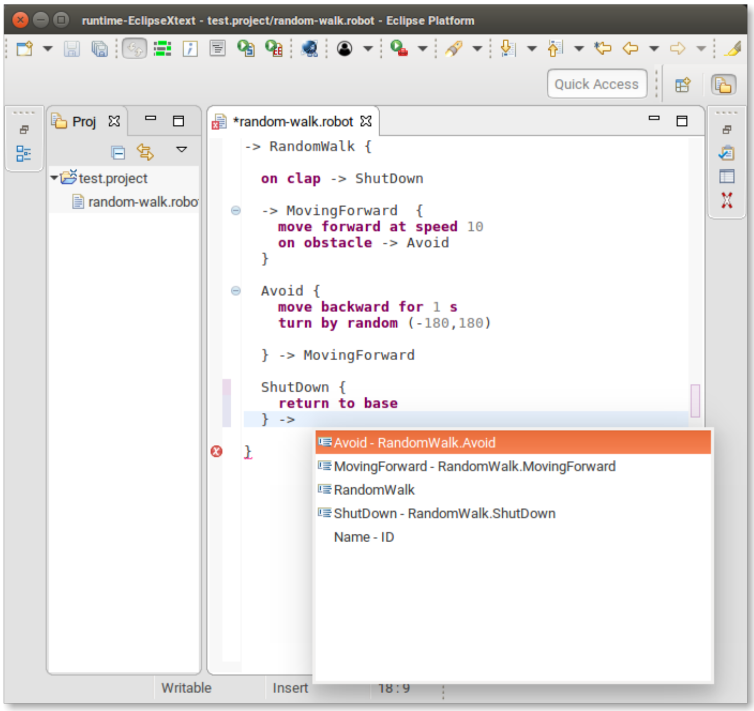
advantages, and disadvantages of various strategies will be discussed in Chapter **??**. For our mobile robot control language we have built the interpreter, whose code is available in the book code repository.

Admittedly, we misuse the adjective *dynamic* when referring to dynamic semantics of DSLs. We follow the terminology developed by the compiler community here. The compiler developers implement programming languages. Their languages describe computation, also referred to as behavior or dynamics. For many DSLs, the semantics will not be dynamic at all, in the sense that they might not be executable. For instance, DSLs for modeling structures (class diagrams), modeling configurations (feature models, see Sect. 8.5.2) or for styling visualizations (CSS) are not dynamic or directly operational. In the end, a CSS style mostly describes how things "look" now how things "behave." In such cases, we basically mean the dynamic semantics to be the implementation of the back-end of the language processing tools, for instance a code generator for class diagrams, a renderer for CSS, and an interactive configurator for feature models.

Tools. Some of the same tools that can be used to specify static semantics can also allow to define dynamic semantics by means of defining operational *reduction rules* in a specialized DSL (see **??**). The reduction rules define an evolution of the system by specifying how an expression specifying a system's state evolves over time through rewriting, somewhat analogously to how re-writing can be used to calculate (evolve) 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 general-purpose programming language. Especially modern functional programming languages such as Haskell, F#, and Scala lend themselves very well for this task. Many of these languages stem from the experience of building the language ML (later known as Standard ML or SML), which was, in fact, developed with meta-programming as its primary use case. The first use and purpose of ML was the implementation of the theorem proving system LCF (Gordon, Milner, and Wadsworth, 1979), where syntax trees of formulae had to be manipulated in proof rules implemented in ML. This is a very similar task to writing interpreters and other language processing tools. We will discuss the implementations of language back-ends in various languages in **??**, **??**, and **??**.

Design environment. A modern programmer appreciates the availability of support tools for the languages used, including rich editing environments with syntax highlighting, error highlighting, code completion, and name resolution. Furthermore, an integration with test infrastructure, for instance, eases working with the language. A screenshot of a generated editor for the mobile robot control language is presented in Fig. 2.8 (produced using the Xtext framework[11] described in Sect. 4.4).

[11] http://www.eclipse.org/Xtext/



```
-> 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
Name - ID
```

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

Specification. The tools used to generate modern development environments depend largely on the specification of syntax and static semantics. Sometimes, additional configuration is required. The editor shown in Fig. 2.8 has been generated solely based on the Xtext grammar specification language, so based on the specifications described above for concrete syntax, abstract syntax, and static semantics. Specifying more custom tools (testers, analyzers, debuggers) is usually not so simple and requires a direct implementation, using similar patterns and tools as for interpreters and generators.

Tools. *Language workbenches* integrate all language implementation components discussed above, and typically add an ability to generate or otherwise create a working integrated development environment (IDE) (Erdweg et al., 2013). A language workbench would normally include a parser generator for handling concrete syntax, along with some facilities for specifying static and dynamic semantics. The workbenches generate editors that combine all the language definition components to provide semantically aware editing (the editor can resolve names, complete references, parse, build 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.

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 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 Chapter 5	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 **????**	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 and **??**	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.

An interesting recent addition to this technologies is the Language Server Protocol (LSP)[12] that allows generic support of rich IDE functionality in any editor and language for which this protocol is implemented. This means that the cost of creating a rich editing experience is dropping dramatically. Once you use a language workbench that can automatically create an LSP server for your language, you obtain a rich experience in any programmer 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. As we advocate moving regular software projects to the MDSE paradigm, we have to admit that testing is also important for implementations of DSLs. Large parts of the logics of

[12] https://langserver.org/

our projects will be embedded in language definitions, and in interpreters and generators. Yet, testing implementations of languages is a bit more complex than testing programs.

Consider the implementation of our robot control language. 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. These are so-called *positive and negative test cases*. 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 robot model in Fig. 2.2 could be used as a positive test case for both syntax and static semantics. The robot 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. Of course, for manual testing, we can create diverse robot models, run the interpreter for each of them, and scrutinize the robot behavior whether it is consistent with the corresponding model. This can be improved slightly, by replacing the physical robot with a simulator and using the, so called, *model-in-the-loop* (MIL) testing. 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 are then becoming triples: a model of a robot, a sequence of inputs, and a sequence of expected actions.

In the remainder of this book, we will be discussing testing patterns in detail for each of the language aspects in the corresponding chapter. We will also define a suitable notion of test cases, discuss what its mean that a test passes or fails (oracles), elaborate on possible stop criteria for testing (for instance, notions of coverage), as well as describe testing architectures and patterns for language implementations.

Further Reading

The basic programming language construction concepts, such as abstract and concrete syntax, parsing, grammars, and left-recursion, are explained in classical compiler text books. If you forgot them, it might be worth keeping one of the standard compiler construction texts (Aho et al., 2006; Mogensen, 2011) as a reference when reading this book.

Fowler and Parsons (2011) give a good, if somewhat traditional, coverage of design issues and implementation techniques for domain-specific languages. 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 (S. Cook et al., 2007), 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 (S. Kelly and Tolvanen, 2008).

Bettini (2013a) 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 idea of language workbenches is usually tracked to an online article by Martin Fowler.[13] They are advocated in detail in the recent book of Voelter (2013), and if you are interested in a good overview on the different features that modern workbenches provide, take a look at a survey paper by Erdweg et al. (2013). As already mentioned, this last paper admits that the technology, under various disguises and various levels of maturity, existed already since the 1980s. Most workbenches were originally designed to facilitate creation of general purpose programming languages, and were adopted over time for designing DSLs. Early workbenches for textual language included: SEM (Teichroew, HERSHEY, et al., 1980), MetaPlex (Chen and Nunamaker, 1989), Metaview (Sorenson, Tremblay, and McAllister, 1988), Centaur (Borras et al., 1988), QuickSpec (Ltd., 1989), MetaEdit (Smolander et al., 1991), Centaur (Borras et al., 1988), ASF+SDF Meta-Environment (Klint, 1993), Gem-Mex/Montages (Anlauff, Kutter, and Pierantonio, 1999), LRC (Kuiper and Saraiva, 1998), and Lisa (Mernik et al., 2002). In recent times, this development has not stopped. Contemporary workbenches for textual languages are JastAdd (Söderberg and Hedin, 2011), Rascal (Klint, Van Der Storm, and Vinju, 2009), Spoofax (Kats and Visser, 2010), Melange (Degueule et al., 2015), and Xtext (Eysholdt and Behrens, 2010; Bettini, 2013a), just to name a few.

Regarding language workbenches for graphical syntax, besides Sirius mentioned above, the more well-known ones include MetaEdit+ (Steven Kelly, Lyytinen, and Rossi, 1996) and several open-source tools building upon the Eclipse Modeling Framework (EMF): the Graphical Modeling Framework (GMF) and Graphiti, which are both part of Eclipse's Graphical Modeling Project.[14]. Other, less known workbenches for graphical languages are DOME (Center, 1999) and GME (Ledeczi et al., 2001).

Projectional editing is not a new idea either. The concept, also called structured or syntax-directed editing, goes back to the 1980s, with tools such as the Incremental Programming Environment (Medina-Mora and Feiler, 1981), GANDALF (Notkin, 1985), and the Synthesizer Generator (Reps and Teitelbaum, 1984). Projectional editing became popular with the Intentional Programming paradigm, which puts language composition at the core of software engineering (Simonyi, 1995; Czarnecki and Eisenecker, 2000). Today, Jetbrains Meta Programming System (MPS)[15] and Intentional's Domain Workbench (Simonyi, Christerson, and Clifford, 2006; Christerson and Kolk, 2009) are the most comprehensive projectional language workbenches.

[13] https://martinfowler.com/articles/languageWorkbench.html
[14] https://www.eclipse.org/modeling/gmp
[15] http://www.jetbrains.com/mps/

Schauss et al. (2017) illustrate many technologies for construction of DSLs. The paper is accompanied by a code repository[16] 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 have sketched an implementation of a very simple external DSL for robot control (a so-called external DSL—implemented as a standalone language). Peterson, Hudak, and Elliott (1999) demonstrate an internal DSL (so, an API-like language), embedded in Haskell for a similar purpose, yet 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 build at higher level of abstraction, aiming for more complex aspects of robots such as reasoning, planning, kinematics, and system architecture. See Nordmann et al. (2016) and its accompanying website[17] for a recent list of more than hundred papers describing robotics DSLs.

Exercises

Exercise 2.1. a) Revisit the Example 5 on page 31. 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 is 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.[18] 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 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

[16] https://softlang.github.io/metalib/

[17] http://corlab.github.io/dslzoo/

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[19] is a deployment and configuration management language. It has started as an internal DSL implemented in Ruby and has grown out into a proper external DSL. Discuss the language based on what you learned about models and DSLs in this chapter, specifically: What is the domain described by models in this language? 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[20] 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, or you can try the same question on other DSLs. For example, if you are interested in robotics, visit the *Robotics DSL Zoo*[21] (Nordmann et al., 2016), pick up one of the languages that attracts your attention, and execute the above discussion for this language.

Note that 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. Be ready to justify your answers briefly. Some possible answers include the interpreter, the type system, regression tests, the code generator, constraints, 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`

[18] Without endorsing any particular vendor, let us name a few that are available as of 2018: Thymio, Lego Mindstorms NXT, Sphero, ArcBotics Sparki, and EdBot. We recommend to browse the web for newer offerings at the time of reading.

[19] https://learn.chef.io/

[20] http://www.slideshare.net/chef-software/overview-of-chef-fundamentals-webinar-series-part-1

[21] http://corlab.github.io/dslzoo/index.html

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

Exercise 2.6. Discuss informally what testing (quality assurance) process would you 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 you would select for each of the constraints? If you have a system where there is many other constraints, and you are time limited in testing, what would be the most important test-cases?

References

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools. Edition 2*. Prentice Hall.

Anlauff, Matthias, Philipp W Kutter, and Alfonso Pierantonio (1999). "Tool support for language design and prototyping with Montages". In: *International Conference on Compiler Construction*. Springer, pp. 296–300.

Backus, J. W., F. L. Bauer, et al. (Jan. 1963). "Revised Report on the Algorithm Language ALGOL 60". In: *Commun. ACM* 6.1. Ed. by P. Naur, pp. 1–17. ISSN: 0001-0782. DOI: 10.1145/366193.366201. URL: http://doi.acm.org/10.1145/366193.366201.

Backus, J. W., R. J. Beeber, et al. (1957). "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, pp. 188–198. DOI: 10.1145/1455567.1455599. URL: http://doi.acm.org/10.1145/1455567.1455599.

Bettini, Lorenzo (2013a). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt.

– (2013b). "Implementing Java-like languages in Xtext with Xsemantics". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, pp. 1559–1564.

Booth, A.D. and K.H.V. Britten (1947). *Coding for the ARC*.

Borras, Patrick et al. (1988). "Centaur: the system". In: *ACM Sigplan Notices* 24.2, pp. 14–24.

Center, Honeywell Technology (1999). *DOME guide*.

Chen, Minder and Jay F Nunamaker (1989). "Metaplex: An integrated environment for organization and information system development". In: *International Conference on Information Systems: Proceedings of the tenth international conference on Information Systems: Boston, Massachusetts, United States*. Vol. 1989.

Chomsky, Noam (1957). *Syntactic Structures*. Mouton & Co.

Christerson, Magnus and Henk Kolk (2009). *Domain Expert DSLs*. talk at QCon London 2009, available at http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk.

COBOL-1961 (1961). *Report to Conference on Data Systems Languages*. Tech. rep. US Dept. of Defense.

Cook, Steve et al. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley.

Czarnecki, Krzysztof and Ulrich W. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley.

Dahl, Ole-Johan and Kristen Nygaard (1967). "Class and subclass declarations". In: *IFIP TC2 Conference on Simulation Programming Languages*.

Degueule, Thomas et al. (2015). "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 2015. Pittsburgh, PA, USA: ACM, pp. 25–36. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814252. URL: http://melange-lang.org.

Erdweg, Sebastian et al. (2013). "The State of the Art in Language Workbenches". In: *SLE*.

Eysholdt, Moritz and Heiko Behrens (2010). "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309.

Felleisen, Matthias, Robert Bruce Findler, and Matthew Flatt (2009). *Semantics Engineering with PLT Redex*. The MIT Press.

Fowler, Martin and Rebecca Parsons (2011). *Domain-Specific Languages*. Addison-Wesley.

Gordon, Michael J. C., Robin Milner, and Christopher P. Wadsworth (1979). *Edinburgh LCF*. Vol. 78. Lecture Notes in Computer Science. Springer. ISBN: 3-540-09724-4. DOI: 10.1007/3-540-09724-4. URL: https://doi.org/10.1007/3-540-09724-4.

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

Hudak, Paul (1996). "Building Domain-Specific Embedded Languages". In: *ACM Comput. Surv.* 28.4es, p. 196. DOI: 10.1145/242224.242477. URL: http://doi.acm.org/10.1145/242224.242477.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, pp. 237–238. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869592. URL: http://doi.acm.org/10.1145/1869542.1869592.

Kelly, S. and J. P. Tolvanen (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley. ISBN: 9780470249253. URL: https://books.google.dk/books?id=GFFtRFkuU%5C_AC.

Kelly, Steven, Kalle Lyytinen, and Matti Rossi (1996). "Metaedit+ a fully configurable multi-user and multi-tool case and came environment". In: *International Conference on Advanced Information Systems Engineering*. Springer, pp. 1–21.

Kleppe, Anneke G. (2009). *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley.

Klint, Paul (1993). "A meta-environment for generating programming environments". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.2, pp. 176–201.

Klint, Paul, Tijs Van Der Storm, and Jurgen Vinju (2009). “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, pp. 168–177.

Kuiper, Matthijs and João Saraiva (1998). “Lrc—a generator for incremental language-oriented tools”. In: *International Conference on Compiler Construction*. Springer, pp. 298–301.

Ledeczi, Akos et al. (2001). “The generic modeling environment”. In: *Workshop on Intelligent Signal Processing, Budapest, Hungary*. Vol. 17, p. 1.

Ltd., Meta Systems (1989). *Quickspec reference guide*.

Matarić, Maja J (2007). *The Robotics Primer*. MIT Press.

McCarthy, John (Apr. 1960). “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4, pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: http://doi.acm.org/10.1145/367177.367199.

Medina-Mora, Raul and Peter H. Feiler (Sept. 1981). “An Incremental Programming Environment”. In: *IEEE Trans. Softw. Eng.* 7.5, pp. 472–482. ISSN: 0098-5589.

Mernik, Marjan et al. (2002). “LISA: An interactive environment for programming language development”. In: *International Conference on Compiler Construction*. Springer, pp. 1–4.

Mogensen, Torben Ægidius (2011). *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, pp. I–XXI, 1–204. ISBN: 978-0-85729-828-7, 978-0-85729-829-4.

Nordmann, Arne et al. (2016). “A Survey on Domain-Specific Modeling and Languages in Robotics”. In: *Journal of Software Engineering in Robotics (JOSER)* 7.1, pp. 75–99.

Notkin, David (May 1985). “The GANDALF Project”. In: *J. Syst. Softw.* 5.2, pp. 91–105. ISSN: 0164-1212.

Object Management Group (2010). *OCL Specification version 2.2*. http://www.omg.org/spec/OCL/2.2/.

Peterson, John, Paul Hudak, and Conal Elliott (1999). “Lambda in Motion: Controlling Robots with Haskell”. In: *Practical Aspects of Declarative Languages, First International Workshop, PADL ’99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*. Ed. by Gopal Gupta. Vol. 1551. Lecture Notes in Computer Science. Springer, pp. 91–105. ISBN: 3-540-65527-1. DOI: 10.1007/3-540-49201-1_7. URL: https://doi.org/10.1007/3-540-49201-1_7.

Reps, Thomas and Tim Teitelbaum (1984). “The synthesizer generator”. In: *ACM Sigplan Notices* 19.5, pp. 42–48.

Rice, H. G. (1953). “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2, pp. 358–366. ISSN: 00029947. URL: http://www.jstor.org/stable/1990888.

Schauss, Simon et al. (2017). “A Chrestomathy of DSL Implementations”. In: *10th International Conference on Software Language Engineering (SLE)*.

Sestoft, Peter (2012). *Programming language concepts*. Springer Science & Business Media.

Simonyi, Charles (1995). “The Death of Computer Languages, the Birth of Intentional Programming”. In: *Proc. NATO Science Committee Conference*.

Simonyi, Charles, Magnus Christerson, and Shane Clifford (2006). “Intentional Software”. In: *Proceedings of OOPSLA*.

Smolander, Kari et al. (1991). “MetaEdit—a flexible graphical environment for methodology modelling”. In: *International Conference on Advanced Information Systems Engineering*. Springer, pp. 168–193.

Söderberg, Emma and Görel Hedin (2011). “Building semantic editors using JastAdd: tool demonstration”. In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, pp. 1–6.

Sorenson, Paul G, J-P Tremblay, and Andrew J McAllister (1988). “The Metaview system for many specification environments”. In: *IEEE software* 5.2, pp. 30–38.

Teichroew, Daniel, EA III HERSHEY, et al. (1980). “Application of the entity-relationship approach to information processing systems modeling”. In:

Universal Robots (Jan. 2015). *The URScript Programming Language. Version 3.1*.

Viyović, Vladimir, Mirjam Maksimović, and Branko Perisić (2014). “Sirius: A rapid development of DSM graphical editor”. In: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, pp. 233–238.

Voelter, Markus (2013). *DSL Engineering. Designing, implementing and using domain specific languages*. URL: http://www.dslbook.org/.

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

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

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 how 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 *meta-modeling 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 (Ernst, 2002).*

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 from 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.

In order to practice meta-modeling, we need a meta-modeling language, often called a meta-meta-model. *A meta-modeling language is a modeling language containing concepts that allow to conveniently describe the abstract syntax of other languages.* Class diagrams are probably the most common meta-modeling language. Especially in the context of MDSE we need expressive and precise meta-modeling languages, so that we can generate the infrastructure for DSLs automatically. However, other languages can also be used, such as feature models Kang et al., 1990, which are also precise, but less expressive than class diagrams (see Chapter 8). Other popular languages used for meta-modeling are XML Schema and other schema languages, as well as the types of most programming languages, in particular algebraic data types in functional languages.

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 page 57) is a class-modeling language popular in meta-modeling in the object-oriented programming community. It is less expressive and simpler than UML class diagrams. Many open-source tools exists around EMF, which we can use to manipulate models and to implement languages. Algebraic data types (or simply ADTs, data types) are typically 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 a relatively powerful tool that is not only used for modeling languages, as is our focus in this book, but it is also used for domain modeling and for model interchange between different tools (Gitzel and Hildenbrand, 2005). For domain-specific modeling see the box on page 53. For model interchange, meta-models are essentially the exchange formats for models. The most prominent standards are the XML Metadata Interchange format XMI (Group, 2015), which is defined by the OMG as part of MOF (see the box on page 57).

3.2 Domain Analysis for Meta-Modeling

DSLs tend to be designed for reasonably mature and 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 shall 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 the 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 the primary purpose, 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 (cf. Sect. 8.5.2), 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 the 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, some overlap and similarity between them appears.

Example 6. Design a language for describing sets of parallel *finite-state machines*. Computer science students will use this language to specify examples and exercises. They need to execute the models in order to interactively explore behaviors (for educational purposes). 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 of adding 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 (Wile, 2004). 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 be described 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 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 Table 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 shall 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 such 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 build 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 shall 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 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 Table 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

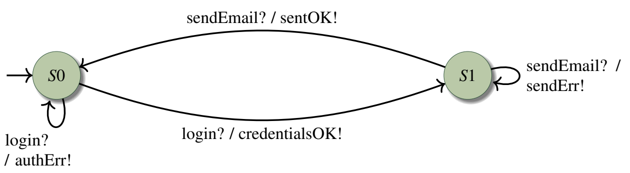
Q1: Purpose	To build examples of students exercises; To interact with examples using an interpreter, an interpreter will be needed.
Q2: Users	Computer science students learning automata theory (probably knowing the basics of a programming language); A professor, who can provide the examples and will 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: 

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

meta-modeling. Once we have a formal meta-model, we will implement our DSL using MDSE. For the state-machine example (Example 6), we 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 Appendix A 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 Table 3.1 when reading. In the figure, we have classes representing finite state machines, states, and transitions. The Model class allows 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

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 http://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 exactly with 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.

we need to store properties (the input and output labels)—associations 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 using the same model 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 property 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 property set to true, 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 property name, we extract it to an abstract class *NamedElement*. This is a common pattern in object-oriented

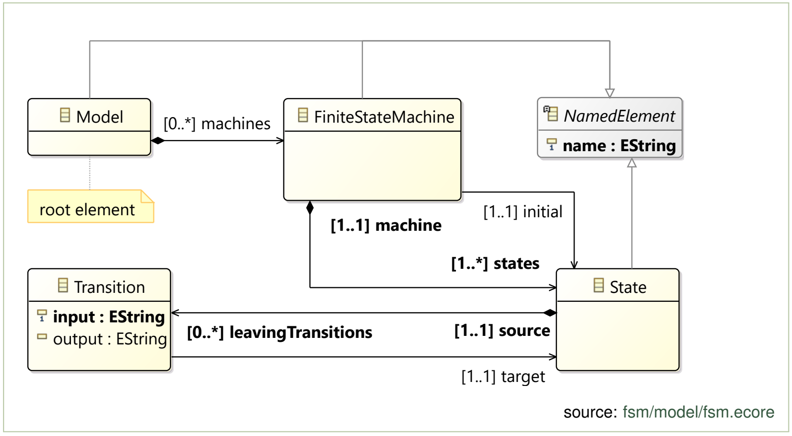


Figure 3.1: A meta-model for the language of finite state machines using class diagrams as the meta-modeling language. Compare with Fig. 5.10.

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

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

Figure 3.2: An example CSS file for Exercise 3.2

meta-models. It allows to use a single visualization code to label objects that are named. For example, the EMF framework itself interprets this property specially, 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 subtree). These can be repeated arbitrary 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` or `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 property of a State, instead being a (non-Boolean) property of the FiniteStateMachine.

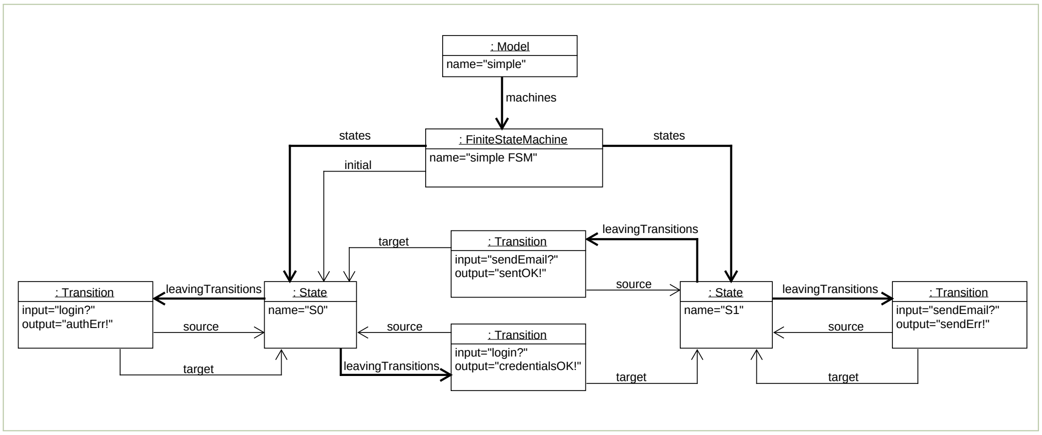


Figure 3.3: *An instance (in abstract syntax) of the finite-state machines language defined by the meta-model in Fig. 3.1*

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.

Instances quickly become large and 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 of how instances and meta-models relate. Figure 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 Table 3.1. We use a UML instance specification diagram to visualize this example.[1] It is instructive to compare the concrete state machine in Table 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 UML instance specification (object diagram) of the meta-model designed in the Exercise 3.2.

[1] Appendix A 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.

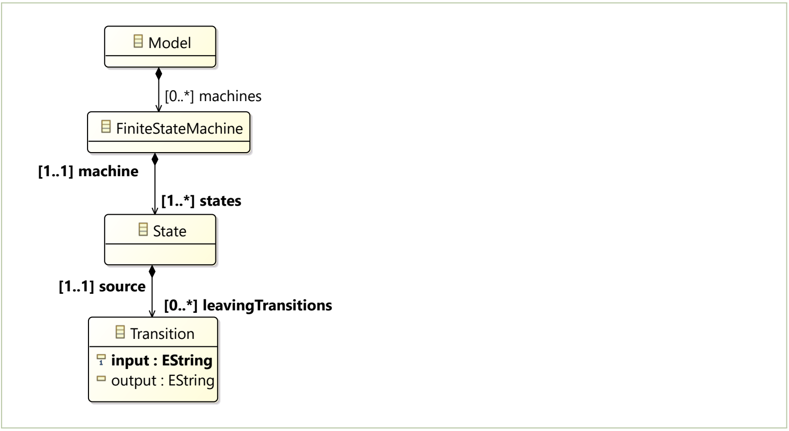


Figure 3.4: *The partonomy of the meta-model of Fig. 3.1*

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.

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.

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 the 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 shall 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 State class (Fig. 3.1). The function should return true if and only if the `this` object is representing an initial state. Not that `isInitial` is a derived property.

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 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, rerunning test-cases, and reduces the risk of manually modifying generated code, which introduces a slippery slope of abandoning all benefits of MDSE in long term.

Finally, the well established Model-View-Controller pattern Krasner, Pope, et al., 1988 calls for separating operations (and visualization) on the model from the model itself. This pattern is commonly followed, and encouraged 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 generalization relation and classes. Unlike for partonomies, quite often, there is no single taxonomy in a meta-model, but several disconnected ones.

> **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 involve the abstract class NamedElement that might not be recognizable for domain experts. 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 the key advantage 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.

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.

Avoid redundancies. It is a bad smell if multiple classes have the same property. If multiple concepts share a property (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 property and the abstract class *NamedElement*.

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

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 to both 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 (2011) 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 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 solution (the so called *Platform Specific Model*). In such case we can use a model-to-model transformation (Chapter **??**) to translate between the problem space model and the solution space model.

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 singular (e.g., Person or Customer), only very rarely in 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 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. Figure 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 will require that all named elements have a string property `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 property—this is, in fact, what EMF does when generating code from our Ecore meta-model.

```scala
trait NamedElement { val name: String }

sealed trait ModelElement

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

type StateName = String

case class FiniteStateMachine (
  name: String,
  states: List[StateName],
  transitions: Map[StateName, List[Transition]],
  initial: StateName
) extends NamedElement with ModelElement

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

source: fsm.scala/src/main/scala/mdsebook/fsm/scala/adt/Pure.scala

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

However, since we are in the pure functional programming setting here, public access to values is much less of an issue than in classical object-oriented programing. Since properties 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 be sometimes 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 anyways hard to evolve language syntax implementation without changing the abstract API, so the external code depending on the API would break anyways 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-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 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. Observe, that a key decision was to lift the transition representation from the state level to the machine level, as a structure over states, not part of the states. The fact that we used a map is somewhat 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 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 very well with pattern matching expressions (switches over types) that allow to concisely write language processing algorithms, especially in interpreters and transformations. Figure 3.6 sketches a skeleton of a type-safe, statically checked code that generically processes any kinds of model elements in our example.

```
me match {
  case Model(name,machines) =>
      ... // code executed if me is an instance of Model
  case FiniteStateMachine(name,states,transitions,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*

```scala
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*

Algebraic data types in functional languages without inheritance (such as Haskell or 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 (type classes, functors, etc.). In languages that combine algebraic data types and classes (such as Scala and F#), most of the meta-modeling forgoes 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 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 marshalling libraries to JSON, YAML, XML, or custom binary formats). If you need a text representation that requires no additional

[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.

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 would 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.

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, has been 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 were 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 fairly concise programming style, well suited for language processing, that comes very useful in later language development stages. The good news is, that this style, with some friction due to imperative nature of its APIs, can be also used with Ecore. Programming languages like Scala, Xtend, and recently also Java, allow to use functional programming 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, 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.

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.

Avoid scope creep. Design what is absolutely necessary and avoid natural tendencies to over-design (Wile, 2004). Regardless 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 that is defined in the language; specifically, question why each construct is needed, and why already now, in the current release of the language. In fact, Wile (2004) suggest to focus on about 80% of the needs, and to provide a way to escape outside the DSL, or to extend it programatically in the underlying system, for the rest of the complex cases. Once the language is created, 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 to run tests with users earlier and thus lowers risks.

Use abstraction wisely. Abstraction is nice, and you should always think what level of abstraction and detail is sufficient or necessary. Consider for

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 (Paige, Kolovos, and Polack, 2013; Krahn, Rumpe, and Völkel, 2010). 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, staying longer on the problem, and avoiding 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 against convoluted types and APIs, and as a result your back-end tools are becoming complex.

instance our FSM language; let us assume that we want to model time in a state machine, expressing execution time. You can decide between having the time in seconds versus just using 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 less 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.

Strive for simplicity. A language must be simple (Kelly and Pohjonen, 2009; Karsai et al., 2009). 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 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 (Bentley, 1986). Few languages never need to be evolved. DSLs are like libraries and need continuous grow. Second, consider making the language open—equip it with some escape mechanism, so that users who outgrow the language, have a possibility to circumvent its limitations (Bentley, 1986). 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 (see **??**), or providing an API to hook into your interpreter or code generator.

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*

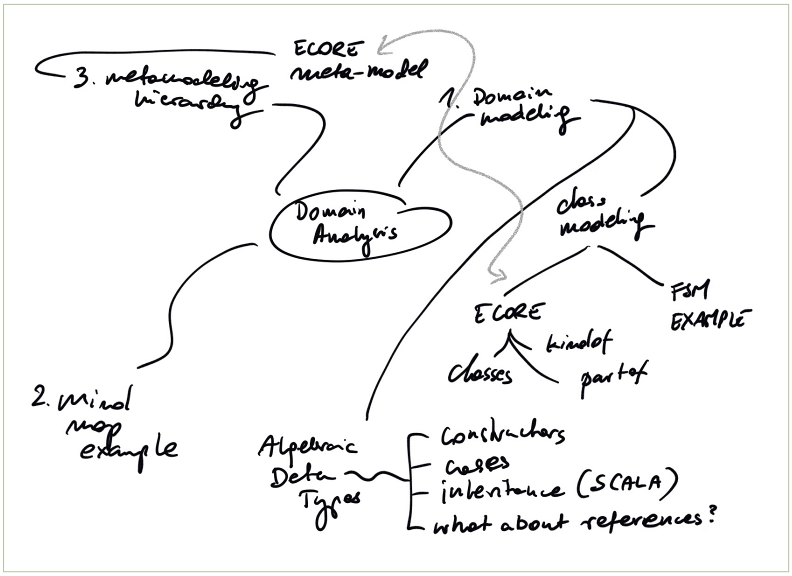


Figure 3.8: An example mind-map, hand-drawn on paper

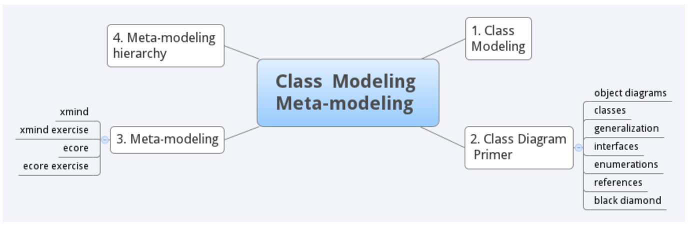


Figure 3.9: A mind-map model shown in concrete syntax (created by the program XMind)

a programming language (Wile, 2004). 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 close as possible (Kelly and Pohjonen, 2009; Stahl and Völter, 2005). 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 (Karsai et al., 2009).

It is better to *use the problem domain as inspiration*, rather than the solution space (Kelly and Pohjonen, 2009). 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 purse a larger example to illustrate the domain modeling and analysis process. Our hypothetical goal is to build a mind-mapping

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 on laptops during exam preparation, or during lectures.
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 neighbours 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 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*

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. Figure 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 would allow 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 Table 3.2. Figure 3.10 shows a potential meta-model of this mind-mapping language. At this point, there should be nothing surprising in this meta-model. Still, let us discuss a specific design decision.

Take a look at the class Color, which we designed to be contained by the class Model. Each topic has optionally a reference to a specific color. We

[3] http://www.xmind.net

[4] http://freemind.sourceforge.net

[5] The example in this section is loosely inspired by a blog post of François Pfister, available at: http://gmf-modeling.blogspot.com, last seen Feb 2016

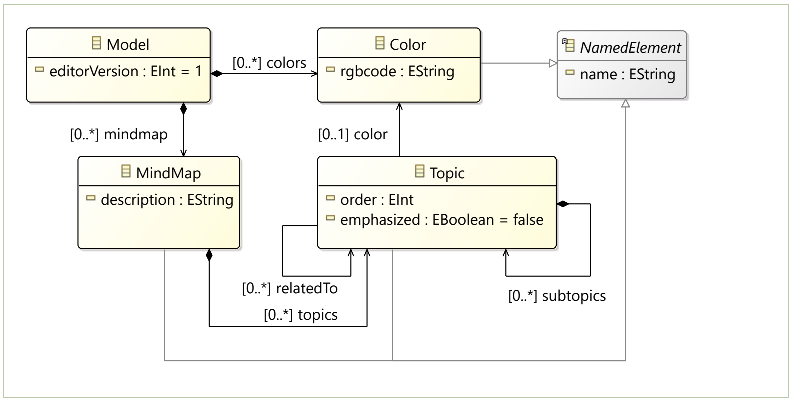


Figure 3.10: *The meta-model of a simple mind-mapping language*

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.

You might also notice another issue with the class Color. When instantiating it, you might find it a bit cumbersome to use it, since you need to create the color code (a string containing three hexadecimal numbers representing the values for the red, green, and blue part of the color). So, you need to put that information into your program code. It might be better to pre-define potential instances of the class Color directly in the model. Unfortunately, that is not possible in Ecore. You could not even create sub-classes of Color and override the attribute rgbcode in the sense of giving it a specific default value, as this is not supported by Ecore. Using constraints (explained in Chapter 5) it could be achieved, but such a solution would be clumsy and still not exactly what you want. In this case, when creating the model, you would still need to manually instantiate the sub-classes of Color, and you could to mistakes here. Instead, what you want is to specify that specific instances will exist in a model. UML in fact has such a construct, called *instance specification*, which we will get back to in Sect. 3.9 (sub-section “Linguistic versus Ontological Instantiation”).

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 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. Conversely, if you work through all the model elements, you will not be able to see if your model misses parts requested in the requirements.

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 (1986) 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 enable separate development and testing of the front-end and the back-end, 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 and so on.

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.*

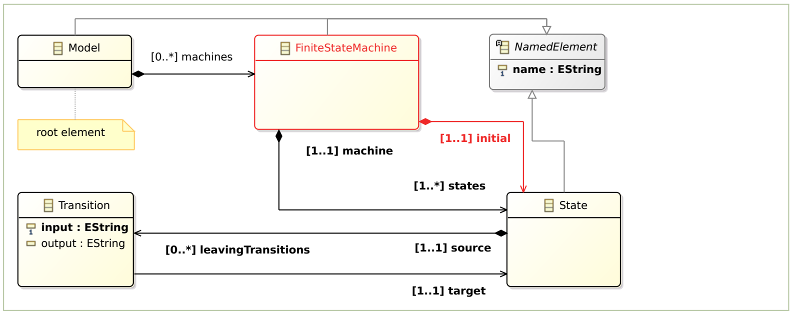


Figure 3.11: *A version of the meta-model for the language of finite state machines that is consistent, but not element-consistent.*

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 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 substantial amount of parent classes, you will need much less minimal instances than all the 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.

If not covered by the examples yet, create a *maximal example* that tries to instantiate all elements that you believe should be possible to instantiate together. 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*, page 68):

Definition 3.5. *A meta-model is* parsimonious *if it contains no meta-classes, no relations (references, associations) and 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 3.6 and 3.4.

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 abstract 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, so explain how would you make sure that the design is satisfactory. What test-cases, or other means, and how many would you use? What are the main properties you want to test a meta-model for?

3.9 The Meta-Modeling 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 into 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.

Let us assume that you are the developer of a new language workbench. The workbench will need to provide the language engineer with some means to create a meta-model, so it offers class diagrams, Ecore, MOF, or some other language that is well-suited to express meta-models (or ADTs). Let us assume you chose Ecore, so you need to implement a tool that your language engineers can use to to create Ecore model. Then the language engineers use the tool to create the Ecore models, an then afterwards they want to generate the language infrastructure for allowing the language users to instantiate the language in terms of a model. The language workbench needs to make sure that this infrastructure supports to only create valid models. Then, finally, in a running system, which is either an interpreter or a code generator, the model is loaded in main memory (more precisely, the heap space) and will be traversed there. So, what we do have is a hierarchy of models at different levels of abstraction, and the models are related via references we call instantiation.

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.

The meta-model of Ecore. In this section, let us first take a look at the top of the hierarchy, which is in the case of many Eclipse-based language workbenches, 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 robotics DSL, the mindmap DSL, or the FSM DSL. These models are then meta-models themselves, since they define all possible instances (models) in our language (e.g., the random walk program from Fig. 2.5 and 2.2 written in our robotics DSL). 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 meta-meta model. In this section, let us first take a look at the top of the hierarchy, which is in the case of many Eclipse-based language workbenches, 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 robotics DSL, the mindmap DSL, or the FSM DSL. These models are then meta-models themselves, since they define all possible instances (models) in our language (e.g., the random walk program from Fig. 2.5 and 2.2 written in our robotics DSL). 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 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 (cf. Appendix B).

The Ecore meta-model of Figure 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. 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,

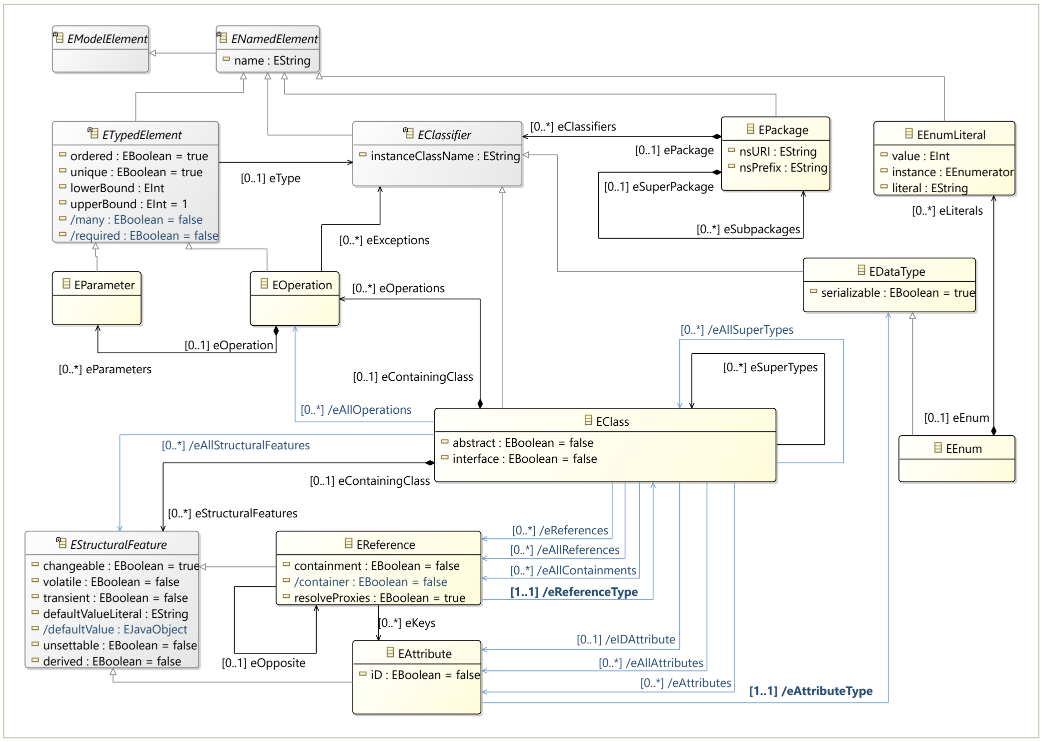


Figure 3.12: An excerpt of the Ecore meta-model of Ecore. In other words: the meta-model of the eCore language, where the meta-model is expressed in Ecore itself.

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 the language in itself could better be called *bootstrapping*. Indeed, it is really akin to the practice of programming language designers, who tend to implement 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 existing compiler (say C). Once this implementation works, one reimplements the interpreter/compiler in Java again, and throws away the temporary C-based 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

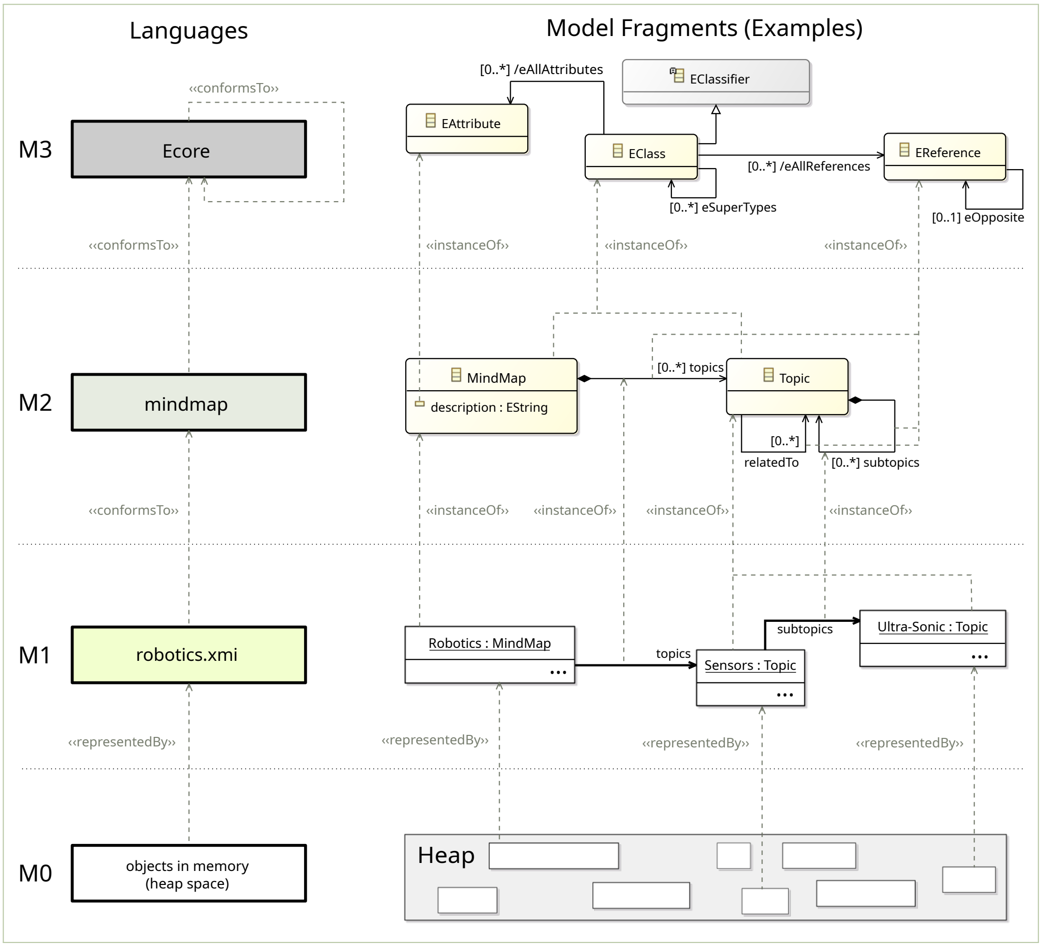


Figure 3.13: Meta-modeling hierarchy illustrated using Ecore and our mind-map example language

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 describing 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* relations 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 will exist in the main memory (when using Java, then in the heap space of the virtual machine) 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.

Note that 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 would have used a concrete syntax, it could look like Fig. 3.8 or Fig. 3.9, depending on how you have designed the concrete syntax.

Figure 3.14 shows the same architecture, but using UML language 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 meta-modeling hierarchy: we have types and instances at the same level, or

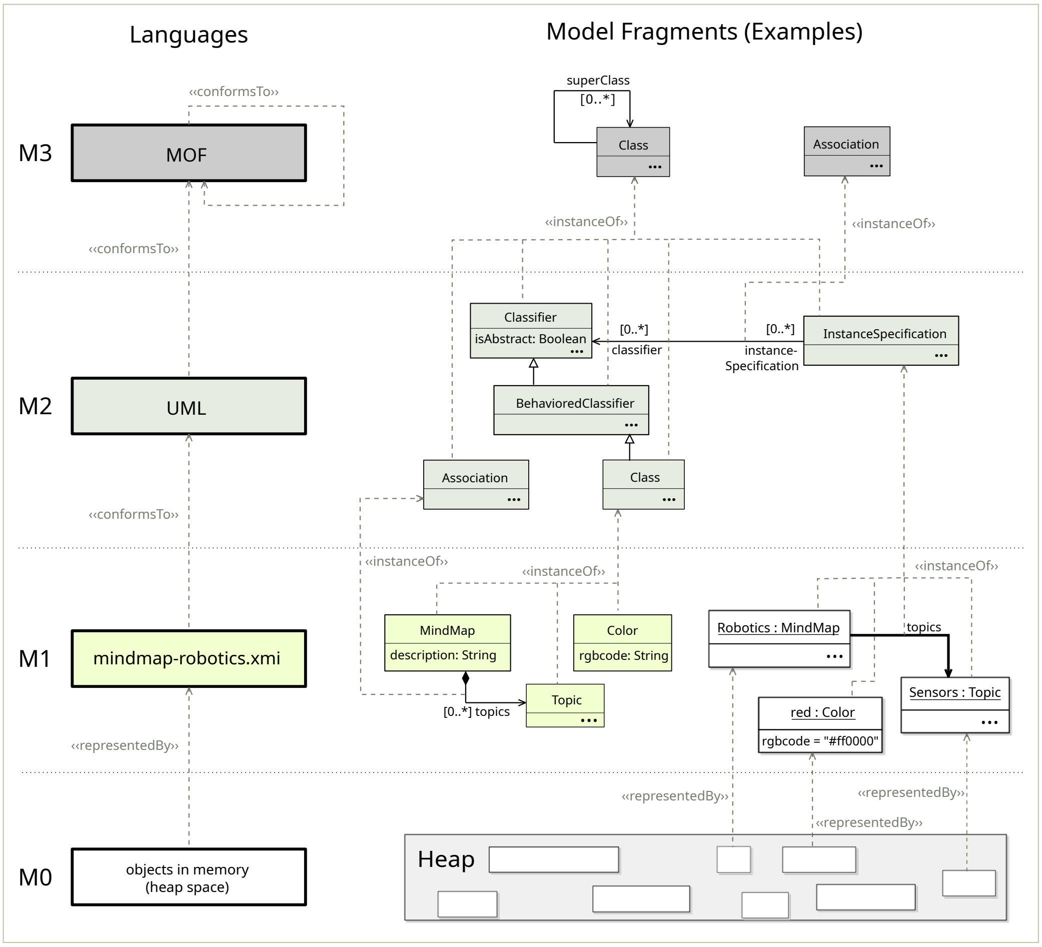
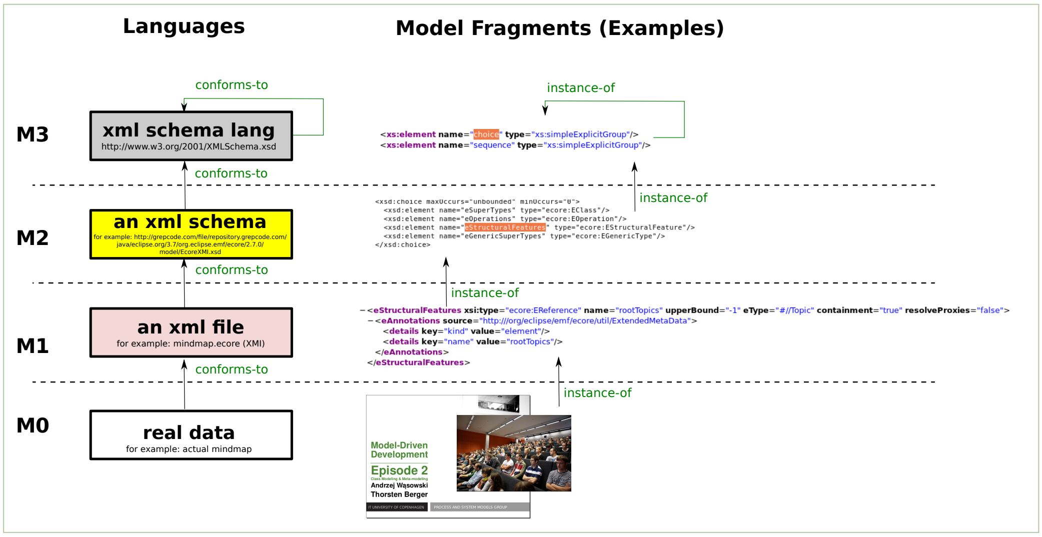


Figure 3.14: Meta-modeling hierarchy illustrated using UML and our mind-map example language

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)


Languages
Model Fragments (Examples)
M3
M2
M1
M0
conforms-to
conforms-to
conforms-to
conforms-to
xml schema lang
http://www.w3.org/2001/XMLSchema.xsd
an xml schema
for example: http://grepcode.com/file/repository.grepcode.com/java/eclipse.org/3.7/org.eclipse.emf/ecore/2.7.0/model/EcoreXMI.xsd
an xml file
for example: mindmap.ecore (XMI)
real data
for example: actual mindmap
instance-of
instance-of
instance-of
instance-of
Model-Driven Development
Episode 2
Andrzej Wąsowski
Thorsten Berger

```
<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 would 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, they do not. If you design your DSL using UML, you will most likely need 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 (Atkinson and Kühne, 2003). The different approaches in their whole are also referred to as linguistic meta-modeling and ontological meta-modeling (Laarman and Kurtev, 2009). 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 (Baader et al., 2003).

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 modeling them together? Programming languages allow specifying both algorithms and values. This need for 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 assume there would only be the object red. Remember that above in Sect. 3.7, we lamented about the missing possibility to pre-define 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 designing more expressive DSLs (Carvalho and Almeida, 2016; Neumayr, Grün, and Schrefl, 2009; Atkinson and Kühne, 2003; Laarman and Kurtev, 2009; Atkinson and Kühne, 2008).

3.10 A Sneak 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.

[6] http://www.w3.org/2001/XMLSchema.xsd

Here, we use the XMI language as an example. At the *M1* level we have 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. The 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 (2011) distinguish domain-models and meta-models, and similarly to this book, they strongly argue for the use of explicit meta-models in the design and implementation of DSLs. Meta-models are called *semantic models* in their book, and they devote an entire section to the pattern of using semantic models in language implementations.

The community advocating MDSE and meta-modeling is commonly referred to as the *modelware community*. However, as we indicated at various opportunities, language development is an old discipline and typically centers around grammars and parsers. That community is commonly called *grammarware community*. For members of that community, it might be difficult to understand the concepts described in this book and used in the modelware community. Paige, Kolovos, and Polack (2013) provide an introduction into meta-modeling concepts for grammarware practitioners. Among others, they motivate the use of a meta-model, which roughly maps to the concept of abstract data types, but is still a relatively unknown concept in the grammarware community.

The Meta-modeling hierarchy is described in section 6.2 of the book by Stahl and Völter (2005). This hierarchy can also be found in the *UML 2.5 Infrastructure Specification* from the Object Management Group. However, the entire specification is not for those of faint heart.

We use the Eclipse Modeling Framework EMF with its language Ecore to illustrate meta-modeling by example. However, technologies evolve quickly and therefore our focus is on the underlying concepts of building DSLs, solely using Ecore for illustrating them. Beyond our quick tutorial on EMF in Appendix B and the scattered descriptions of EMF specifics in this book, we recommend the following books to gain in-depth knowledge into EMF: Steinberg et al. (2009), Budinsky et al. (2004), and Moore et al. (2004).

If you want to hear some background about the origin of EMF and Ecore, together with some personal reflection on it, it is probably worth reading an interview with Ed Merks, one of the initiators and main designers of EMF and Ecore at: https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html.

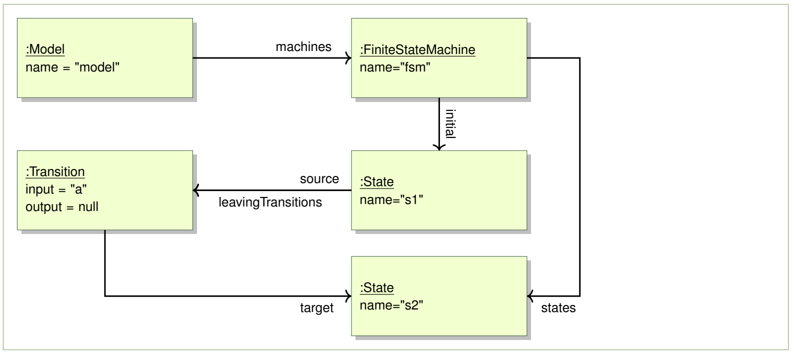


Figure 3.16: An example invalid instance of the FSM meta-model of Fig. 3.1

Finally, as with many technologies, we recommend poking online, such as at the respective websites of Wikipedia,[7] Object Management Group,[8] and EMF[9] to obtain more details about Meta-Modeling, EMF, Ecore, MOF, XMI, and about the relations between these concepts.

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

Exercise 3.15. Figure 3.17 presents a simplified meta-model for SQL queries. Draw an 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;`

[7] https://en.wikipedia.org/wiki/Metamodeling
[8] http://www.omg.org/mof
[9] http://www.eclipse.org/modeling/emf/

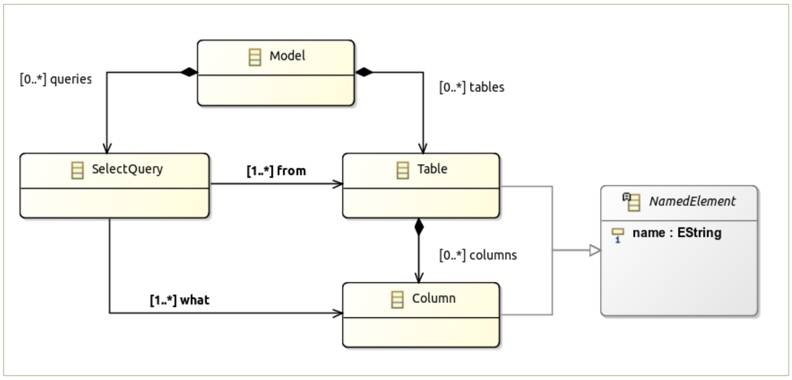


Figure 3.17: *A simple meta-model for SQL queries*

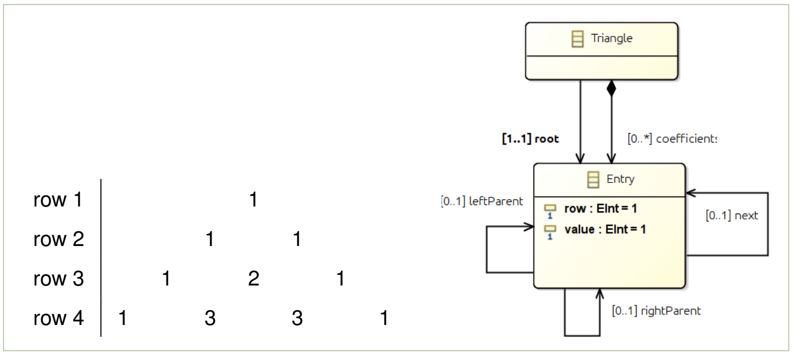


Figure 3.18: *A meta-model for Pascal's triangle*

b) `SELECT NAME, PRICE FROM PRODUCT;`

Exercise 3.16. Pascal's triangle (see the left part of Fig. 3.18) is a numeric hierarchical structure, where each internal node's value is a sum of the values of its two parents, one row above.

The right part of Fig. 3.18 presents a meta-model to represent Pascal's triangles of different sizes. All numbers are stored in entries nested directly under an instance of the root class called `Triangle`. Then additional references are used to connect nodes to parents and to their next sequential neighbour.

Draw the abstract syntax of the triangle in the left part of the figure as an instance of the Ecore meta-model in the right side of the figure. To save the time, only draw the instance for the first three rows (ignore row 4).

Exercise 3.17. Consider the feature model in concrete syntax shown in Fig. 3.19. Figure 3.20 shows a simplified meta-model for this modeling language. Draw the abstract syntax of the above feature model as an instance of this meta-model. In concrete syntax we draw a hollow arc to denote `XorGroup` and a filled on to denote `OrGroup` members. There is only an Xor-group in the instance. Remember that an abstract syntax tree must have a single partonomy. More information about feature models is available in Chapter 8.

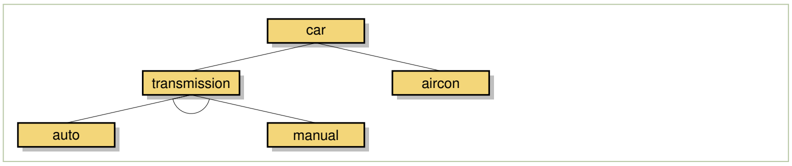


Figure 3.19: A simple feature model in concrete syntax

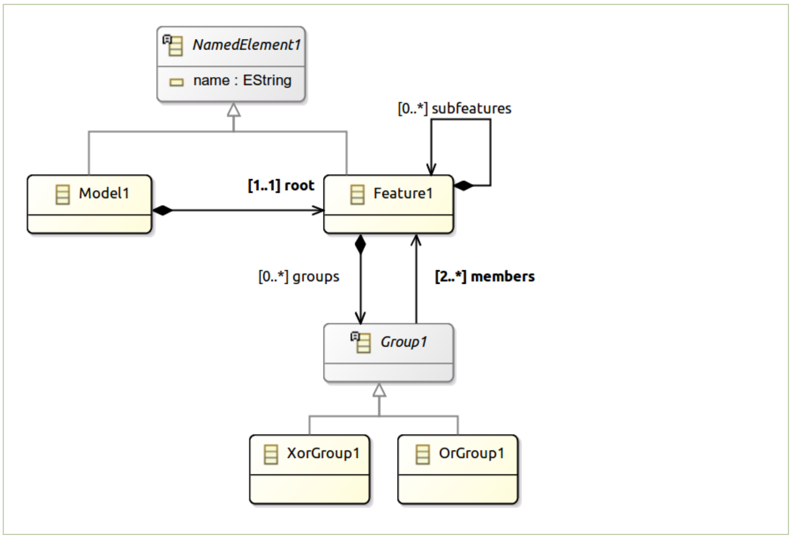


Figure 3.20: A meta-model for feature diagrams

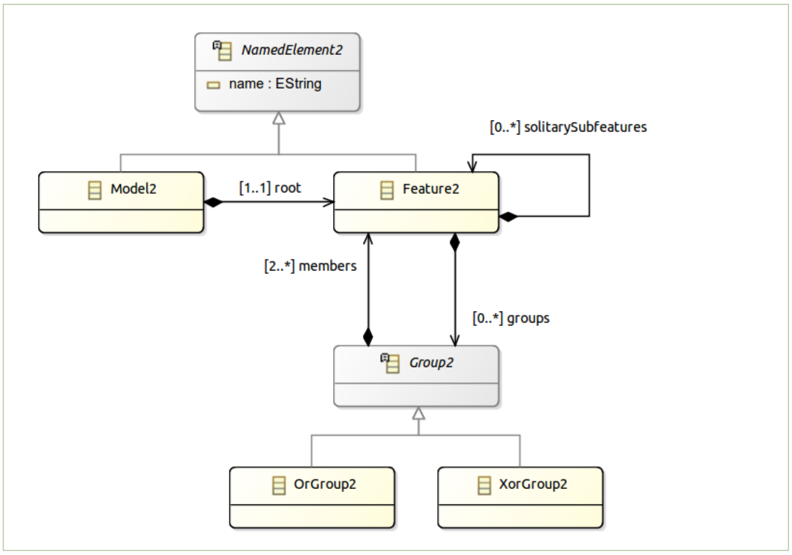


Figure 3.21: An alternative meta-model for feature-models

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.

```
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*

Exercise 3.19. A simple meta-model for feature diagrams is presented in Fig. 3.20.[10] Extend this meta-model 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 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 representing optional features. 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 a property '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 divions (`div`).

Design a meta-model representing documents in this subset of HTML. Consider adding anchors and anchor references to your meta-model. The exercise makes sense if you 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 labelled 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 (Chapter 1.2). 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 property 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. Consider the following domain: describing very simple class specifications. Each class has a name; a class can be abstract or concrete; and a

[10]Feature models are discussed in detail in Chapter 8

```
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*

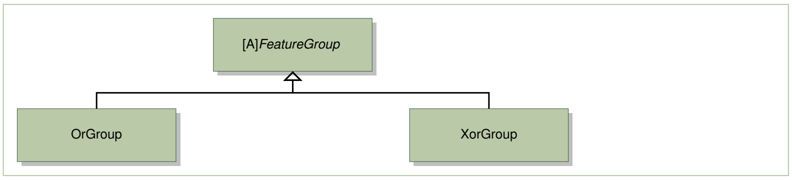


Figure 3.24: *A simple class diagram*

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).

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 simplified XML dialect. Note, that this is meant to be done for XML as a language, not for a particular XML dialect.

Exercise 3.27. Now, design a meta-model (a schema!) for an XML dialect known to you. Explain the main difference between this meta-model and the meta-model of 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. Consider the simple class-diagram in Fig. 3.24, which (incidentally) shows a fragment of a meta-model for feature diagrams. Draw the abstract syntax tree of this diagram as an instance of the simplified Ecore presented 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.

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

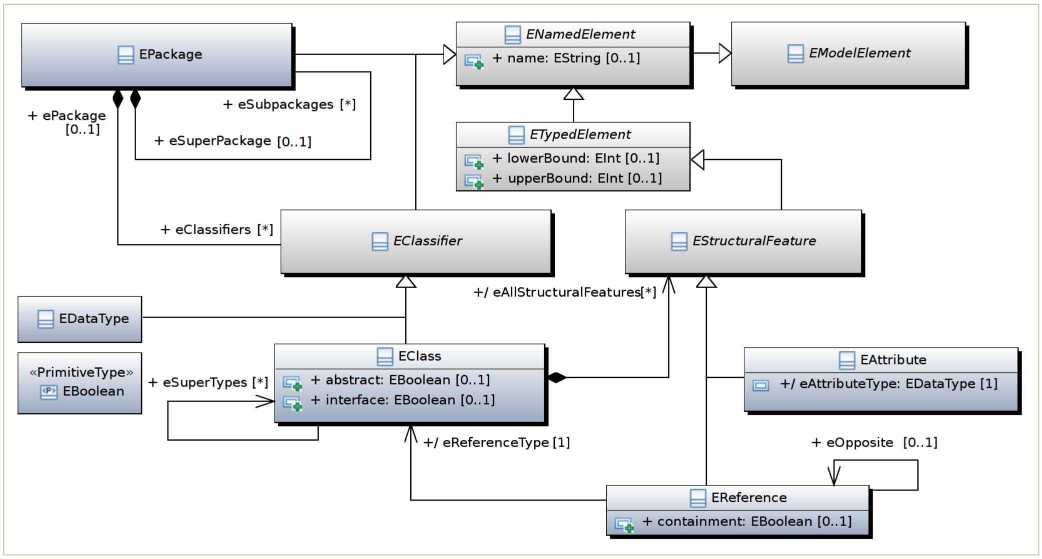


Figure 3.25: A subset of the Ecore meta-model

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

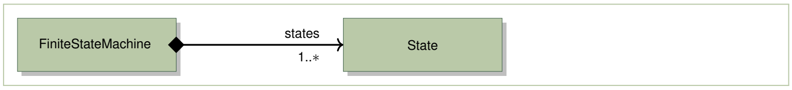
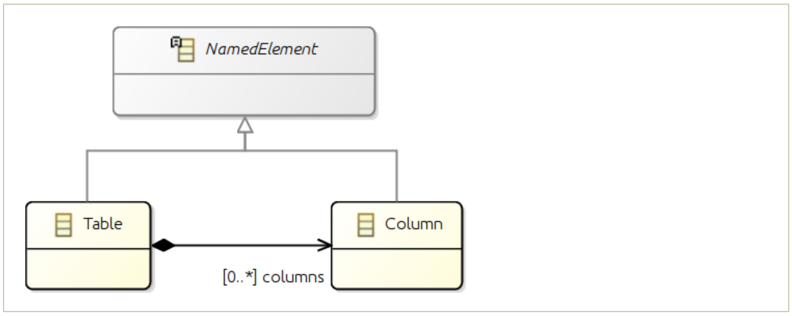


Figure 3.27: A tiny meta-model for relational data (entity-relationships diagrams)



Only show the part of the abstract syntax that partains to what is in Fig. 3.27 (so ignore that it could be contained in a package, etc).

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 obtaining a kind of boot-strapping. We attempt to understand this idea on a small part of Fig. 3.25. Consider the part 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. In the abstract syntax include classes, generalizations, references, and properties such as abstract, containment, and names.

Only show the part of the abstract syntax that pertains to what is in Fig. 3.28 (so ignore that it could be contained in a package, have other properties, cardinalities, etc.).

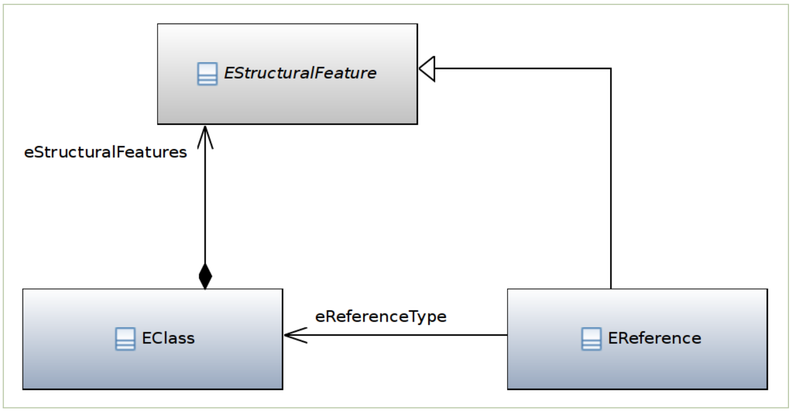


Figure 3.28: *A fragment of the Ecore meta-model*

Exercise 3.33. Recall that Ecore supports bidirectional associations only indirectly (see Appendix A.4). 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 abstract syntax for a simple object diagram showing two classes related by a bidirectional references.

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 page 185. See also Exercise 4.57 on page 150.

References

Atkinson, Colin and Thomas Kühne (2003). “Model-driven development: a meta-modeling foundation”. In: *IEEE software* 20.5, pp. 36–41.
– (2008). “Reducing accidental complexity in domain models”. In: *Software & Systems Modeling* 7.3, pp. 345–359.
Baader, Franz et al. (2003). *The description logic handbook: theory, implementation and applications*. Cambridge University Press.
Bentley, Jon (Aug. 1986). “Programming Pearls: Little Languages”. In: *Commun. ACM* 29.8, pp. 711–721. ISSN: 0001-0782. DOI: 10.1145/6424.315691. URL: http://doi.acm.org/10.1145/6424.315691.
Budinsky, Frank et al. (2004). *Eclipse Modeling Framework*. Addison-Wesley.
Carvalho, Victorio A. and João Paulo A. Almeida (June 2016). “Toward a well-founded theory for multi-level conceptual modeling”. In: *Software & Systems Modeling*. ISSN: 1619-1374.
Ernst, Johannes (2002). *What is metamodeling and what is it good for?* http://infogrid.org/wiki/Reference/WhatIsMetaModeling.
Fowler, Martin and Rebecca Parsons (2011). *Domain-Specific Languages*. Addison-Wesley.
Gitzel, Ralf and Tobias Hildenbrand (Apr. 2005). *A Taxonomy of Metamodel Hierarchies*. URL: https://madoc.bib.uni-mannheim.de/993/.
Group, Object Management (2015). *Metadata Interchange (XMI) Specification*. URL: https://www.omg.org/spec/XMI.
Kang et al. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21.

Karsai, Gabor et al. (2009). “Design Guidelines for Domain Specific Languages”. In: *9th OOPSLA Workshop on Domain-Specific Modeling*. URL: http://www.dsmforum.org/events/DSM09/Papers/Karsai.pdf.

Kelly, Steven and Risto Pohjonen (2009). “Worst Practices for Domain-Specific Modeling”. In: *IEEE Software* 26.4, pp. 22–29.

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

Krasner, Glenn E, Stephen T Pope, et al. (1988). “A description of the model-view-controller user interface paradigm in the smalltalk-80 system”. In: *Journal of object oriented programming* 1.3, pp. 26–49.

Laarman, Alfons and Ivan Kurtev (2009). “Ontological metamodeling with explicit instantiation”. In: *International Conference on Software Language Engineering (SLE)*.

Moore, Bill et al. (2004). *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM Redbooks.

Neumayr, Bernd, Katharina Grün, and Michael Schrefl (2009). “Multi-level Domain Modeling with M-objects and M-relationships”. In: *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*. APCCM ’09.

Paige, Richard F, Dimitrios S Kolovos, and Fiona AC Polack (2013). “Metamodelling for Grammarware Researchers”. In: *5th International Conference on Software Language Engineering (SLE)*.

Stahl, Thomas and Markus Völter (2005). *Model-Driven Software Development*. Wiley.

Steinberg, David et al. (2009). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.

Wile, David S. (2004). “Lessons learned from real DSL experiments”. In: *Sci. Comput. Program.* 51.3, pp. 265–290.

Parser development
is still a black art
(Klint, Lämmel, and Verhoef, 2005)

4 Concrete Syntax

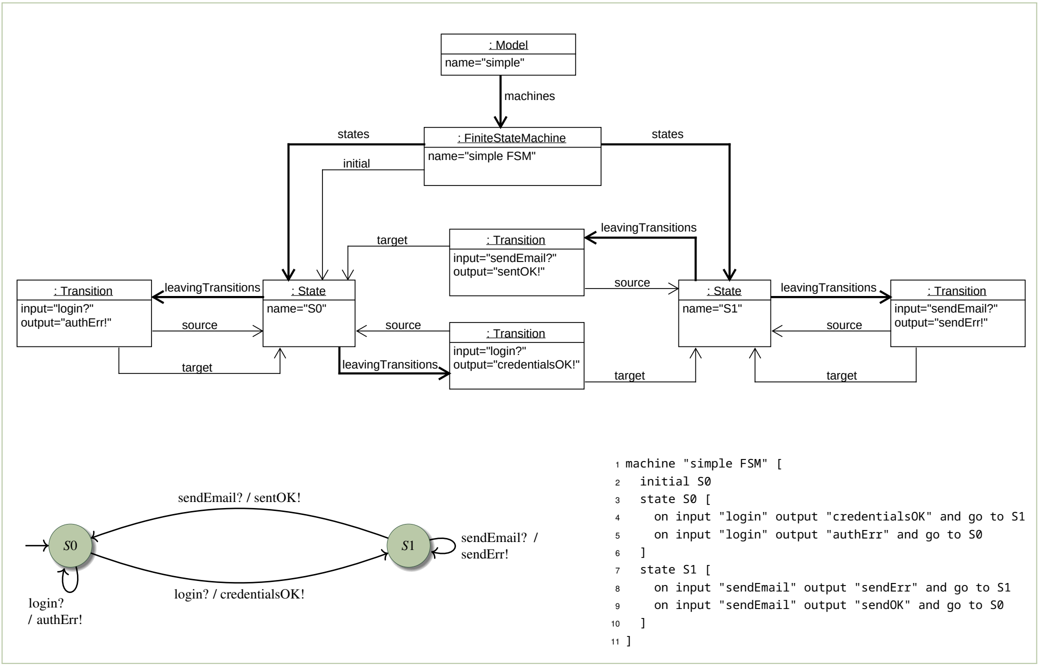
Models and meta-models, algebraic data types and values, XML schema and files, class and instance diagrams, YAML files—all these abstract syntax specification methods are clearly important for you as a language designer. At the same time, the end users, especially domain experts who are not programmers, 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 the language users, remaining 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. Graphical syntax is the subject of **??**. However, we try to limit the theoretical considerations to bare minimum. We hope that with this chapter we can actually meet the wishes of Klint, Lämmel, and Verhoef (2005) expressed in the paper quoted on the top of this page: demystify creation of parsers, showing and systematizing how grammars are written. The chapter contains examples, case studies, 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 Table 3.1. Meant for human consumption, it uses graphical elements (e.g. arrows) on top of 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 brings the very same model in a *textual concrete syntax*, a text-based representation aiming 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:



```
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 in concrete textual syntax (bottom right). Convince yourself that the three representations indeed capture the same model

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 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 encode 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 a line between is placed between 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

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 distinguished 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 if 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.

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

Exercise 4.1. Revisit the examples of 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 the correct programs from the incorrect ones, and to *translate* them to 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 page 6.

The software language engineering community agrees that the 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, this set of characters is typically a variant of Unicode. However, since regular expressions can be used to describe other things than program text, let us just 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 a syntax of a binary number. For this example, we take the alphabet to be all letters and digits:

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

although we will only use the first 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 deriving many interesting constructs (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):

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

Even with the alternative operator, we can only describe finite numbers of tokens. Worse, our regular expressions are as large the languages they describe. We cannot possible list 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 with 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:

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

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 (alternative or 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):

$$
\begin{aligned}
[\![\varepsilon]\!] &= \{\varepsilon\} \\
[\![a]\!] &= \{a\} \qquad \textit{for any } a \in \Sigma
\end{aligned}
$$

Inductive case (composite expressions):

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

The first inductive case in Def. 4.13 says that the generated language contains any word generated either by r or 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 finite automaton. If you are interested more about its theoretical properties, please refer to a more foundational text on theory of automata (for example Hopcroft, Motwani, and Ullman (2001)). 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 expressions operators introduced above. *Positive examples:* 0, 10, 11, 10101011; *Negative examples:* 00, 01, 011, 0000000

Table 4.1 lists several examples of 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 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 shall facilitate translation from textual input to abstract syntax

[1] See for instance https://www.regular-expressions.info/posixbrackets.html, accessed 2019/08

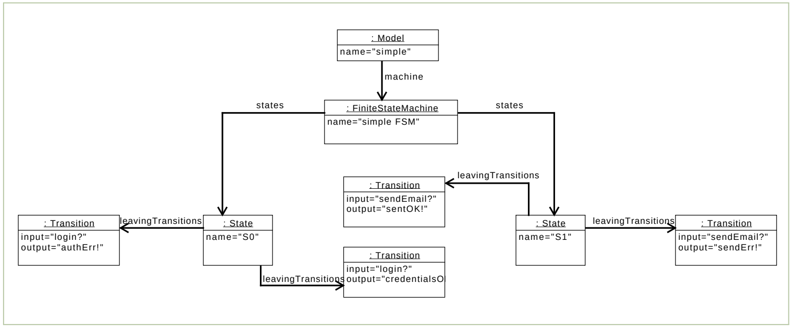


Figure 4.2: *A decomposition of the abstract syntax instance shown in the top of Fig. 4.1. Convince yourself that it is an instance of the partonomy of Fig. 3.4*

trees. Yes, the core structure of any abstract syntax representation is a tree (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, a model, or a program. 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, 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 to describe such an inductive generation of trees. Consider the example below.

Example 12. Let us develop an intuition how grammars capture program text by analyzing syntax of arithmetic expression, 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:

Operator name	**Expression**	**Expansion**
optional	$r?$	$r\|\varepsilon$
Kleene star	r^*	$r^+\|\varepsilon$
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*

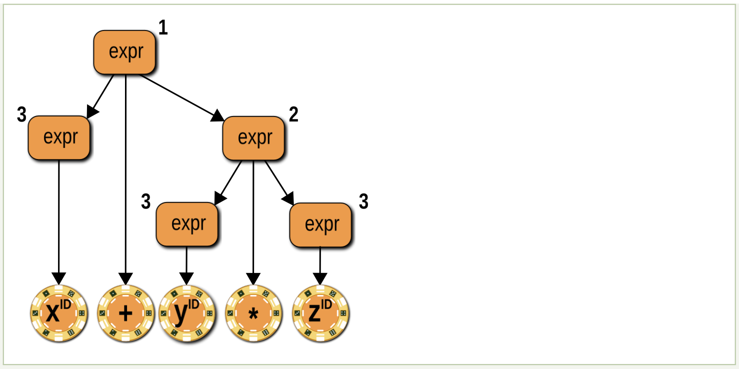


Figure 4.3: An abstract syntax tree for the expression $x+y*z$, an informal notation. Numeric labels indicate a possible derivation order, explained below

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 should we 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]

$$\mathsf{expr} \rightarrow_3 \ \mathtt{ID} \ , \tag{4.14}$$

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-subexpression, then an operator token, and a right subexpression:

$$\mathsf{expr} \rightarrow_2 \ \mathsf{expr} \ \ \mathtt{'*'} \ \ \mathsf{expr} \tag{4.15}$$

$$\mathsf{expr} \rightarrow_1 \ \mathsf{expr} \ \ \mathtt{'+'} \ \ \mathsf{expr} \tag{4.16}$$

Note how this structure with nesting subexpressions (instead of identifiers directly) allows us to represents 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):

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

In the above, we always expand the rightmost occurrence of expr using one of our rules, as labelled 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 right-most 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:*

$$n \rightarrow \sigma \quad .$$

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 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 do they define, is explained in the semantics of context-free grammars:

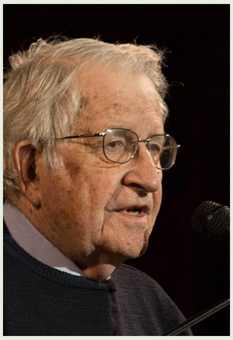
Definition 4.19 (Semantics of Context-Free Grammars)**.** *Assume that s is a start non-terminal of a context-free grammar G, with a production relation $\rightarrow \subseteq N \times T$. Then the grammar G* generates *a language of words (sequences) over the alphabet of terminals T as follows:*

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

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

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

The Unusual Past of Formal Grammars



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 *Syntactic Structures* (Chomsky, 1957).

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.

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 sequence, without taking into consideration its context. In Example 4.30, 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:

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

Add two terminals, representing 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 the following context-free grammar (small letters denote non-terminals, quoted letters terminals, ε the empty string, and n is the start symbol):

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

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, are overlapping significantly. As noted in Chomsky's Hierarchy (info box on p. 103), 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. 99) 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 shall 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 (Table 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? Foremostly, 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 Table 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 Actually Write a Grammar in Practice?

Having seen the basic specification notations, we want to understand how these 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 invariably brought by tools. A *base-line grammar* is best created using fundamental notations (Klint, Lämmel, and Verhoef, 2005), 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 how 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. the last row of Table 3.1).

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

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*

```
1  let State = { S0, S1 }
2
3  let Tran = {
4    transition (S0, "login"?, "credentialsOK"!, S1)
5    transition (S0, "login"?, no!, S0)
6    transition (S1, "sendEmail"?, "sendErr"!, S1)
7    transition (S1, "sendEmail"?, "sendOK"!, S0)
8  }
9
10 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 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 finally 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.

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 (Table 3.1) states cannot be nested, this syntax will 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 underspeci-

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`
State 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 to quote 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

fied 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 initial language requirements. For the FSM example, we extract the requirements and issues from the domain analysis in Table 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 Table 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.

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:

The above three categories of tokens are typical for most languages. Punctuation can be further divided into separators (comma, colon, semi-colon), operators (plus, minus, navigation dot), and delimiters (parentheses,

```
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*

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.*

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 0 constant 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 (mostly punctuation and keywords) we create a terminal symbol representing it. With most tools this happens automatically—token definitions are made available as grammar symbols. For convenience, we typically do not use these terminals explicitly, but write regular expressions directly in the grammar, as most tools allow this. Table 4.4 lists the terminal symbols of our grammar in the right-most column.

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 be later 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 to an integer value.

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 Table 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 to prioritize tokens, 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, like 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. 31). Formulate regular expressions for the identified categories of tokens. The exercise should result in a table similar to Table 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 to discover some of these, 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 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, this declaration 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`–`S6`), 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. Figs. 3.1 and 4.5

syntactic categories with the level of detail of Table 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 can see the syntactic categories of your language, writing the grammar productions is quite easy. Syntactic categories become nonterminals, tokens become terminal, and your syntax specification governs the rules (mostly try to cover the examples you generated by now). We begin with several simple rules from the bottom of Table 4.5:

$$\begin{aligned}
\textsf{inputClause} &\rightarrow \texttt{'on'}\ \texttt{'input'}?\ \texttt{Id} \\
\textsf{outputClause} &\rightarrow \texttt{'output'}\ \texttt{Id}\ \texttt{'and'} \\
\textsf{targetClause} &\rightarrow \texttt{'go'}\ \texttt{'to'}\ \texttt{Id} \\
\textsf{transition} &\rightarrow \textsf{inputClause}\ \ \textsf{outputClause}?\ \ \textsf{targetClause} \\
\textsf{stateBlock} &\rightarrow \texttt{'state'}\ \texttt{Id}\ (\ \texttt{'['}\ \textsf{transition}^*\ \texttt{']'})? \\
\textsf{initialDeclaration} &\rightarrow \texttt{initial}\ \texttt{Id}
\end{aligned} \tag{4.21}$$

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 are up for 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, for instance, 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, or after the initial declaration. Now that this piece of grammar becomes large enough to give it a non-terminal name:

$$\begin{aligned}\mathsf{machineBlockContents}\ \rightarrow\ & \big(\ \mathsf{stateBlock}^{+}\ \ \mathsf{initialDeclaration}\ \ \mathsf{stateBlock}^{*}\big)\\ & \mid \big(\ \mathsf{stateBlock}^{*}\ \ \mathsf{initialDeclaration}\ \ \mathsf{stateBlock}^{+}\big)\end{aligned} \tag{4.23}$$

Defining the shape of allowed inputs precisely quickly becomes quite cumbersome. It is typically better to settle with simple, approximating presentations like Eq. (4.22). It 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:

$$\begin{aligned}\mathsf{machineBlock}\ &\rightarrow\ \texttt{'machine'}\ \ \texttt{Id}\ \big(\ \texttt{'['}\ \ \mathsf{machineBlockContents}?\ \texttt{']'}\big)?\\ \mathsf{model}\ &\rightarrow\ \mathsf{machineBlock}^{*}\end{aligned} \tag{4.24}$$

We conclude the section, by summarizing the 6-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.

```
grammar mdsebook.fsm.xtext.Fsm
    with org.eclipse.xtext.common.Terminals
import "http://www.mdsebook.org/mdsebook.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/mdsebook/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

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 erronous 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 encouraged you to think about grammars as generators of legal models and programs in the language. A parser does the opposite: it checks (recognizes) if a given input model belongs to the language; whether it could have 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` in 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). However, these tools need more information than a plain context-free grammar provides. EBNF just describes how to structure input symbols intro 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 the 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 language specification using 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 in the end of the chapter). Parsing is a very specialized field of knowledge by itself. 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] 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 in 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 modularizing and reusing grammars. In particular reusing the specification of terminals is very useful as most modern languages share vast majority of terminals (string literals, numeric literals, identifiers, and operators). Lines 3–4 import the meta-models defining the abstract syntax. We are importing the FSM meta-model

[3] https://www.eclipse.org/Xtext/

[4] https://www.antlr.org/

[5] https://langserver.org/, more on this topic in Chapter 7

(Fig. 3.1) and Ecore. These two imports will allow using the types of abstract syntax in the grammar productions 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 `mdsebook.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 subtype of the type specified in the `returns` clause. Xtext will translate this action to 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 constructed objects will be added to the `machines` collection of the returned `Model` object. Check Fig. 3.1 on p. 58 to convince yourself that a `Model` object indeed has a property `machines`, and that this property is indeed a collection (it has 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 to 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 identifer 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 is not constructing 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 $\mathtt{T} \to \alpha\ (\beta\ \&\ \gamma)\ \delta$ 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 subexpressions in EBNF. You may want to use Table 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 to 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 is already 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 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. Appendix C presents a short tutorial to using the Xtext framework, if you are interested in learning more about this tool.

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 why Parboiled2 is very fast, unlike most parser combinator libraries. We will talk more about internal DSLs in **??** and mechanisms for meta-programming in **??**.

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 (Ford, 2004). 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

[6] https://github.com/sirthias/parboiled2

```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/mdsebook/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 core part with non-terminal productions is shown here*

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. The 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 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 PEG in Parboiled2 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. In the figure, 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 parser combinator libraries will look similarly.

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 rules 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 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 actions 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

```
1   def STRING: Rule1[String] =
2     rule { WS.? ~ '"' ~ capture ((!'"' ~ ANY).*) ~ '"' ~ WS.? }
3
4   val IDFirst: CharPredicate = CharPredicate.Alpha ++ "_"
5   val IDSuffix: CharPredicate = CharPredicate.AlphaNum ++ "_"
6
7   def ID: Rule1[String] =
8     rule { WS.? ~ capture (IDFirst ~ IDSuffix.*) ~ WS.? }
9
10  def BEGIN =
11    rule { WS.? ~ '[' ~ WS.? }
12  def END =
13    rule { WS.? ~ ']' ~ WS.? }
14
15  implicit def StringWS (s: String): Rule0 =
16    rule { WS.? ~ str (s) ~ WS }
17
18  def WS: Rule0 =
19    rule { anyOf (" \n\t").+ }
```

source: fsm.scala/src/main/scala/mdsebook/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*

they can be used to define the lexical structure of the language as well. This is typically the approach taken by PEGs 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 character 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 productions (`Rule0`). `Rule0` is the type of rules that do not return any interesting value, but consume 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.

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 (1989), 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 of expressing 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 this semantics are easier to understand to users than the alternatives (say shift-reduce parsing).

Over time, parser combinators libraries became a part of the basic infrastructure of any serious programming language: Java, Scala (Parboiled, Parboiled2, and Petit Parsers), JavaScript (Bennu, Parjs, Parsimmon), C# (pidgin, superpower, parseq), C++ (Cpp-peglib, boost meta-parse, boost-spirit, ParserCombinators), Python (Parsec.py, Parsy, Pyparsing, parsita) and so on.

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/mdsebook/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 `mdsebook.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 **????**.

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 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 below in Fig. 4.9:

```scala
val Transition: (String,Option[String],String)=>Pure.Transition =
  (input, output, target) =>
    Pure.Transition (target, input, output getOrElse "")
```

source: fsm.scala/src/main/scala/mdsebook/fsm/scala/FsmParser.scala

Figure 4.9: *The semantic action for constructing instances of transition*

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. These general way to specify semantic actions is far more expressive than the constructor calls and property assignments of Xtext. It almost does not happen 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 pre-defined 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 much harder development experience. In the words of Ford (2004), *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 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).

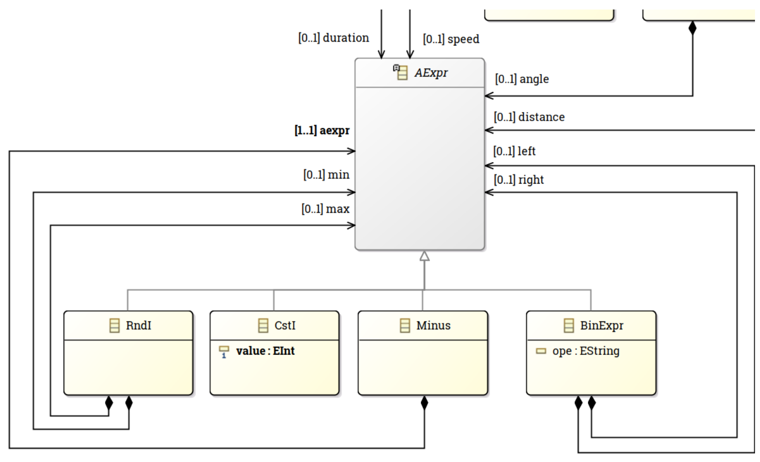
Consider not writing a grammar. No parser at all! Standard format technologies (YAML, JSON, XML, and CSV)[7] are natural alternatives to bespoke syntax. They allow fast and ad hoc creation of file formats with efficient parsers and validity checkers. These are excellent for many structural and configuration modeling tasks. On the other hand, bespoke syntax may be needed if humans have to create models in an editor, when 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 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 worth to invest in creating multiple front-ends for the same abstract syntax, to allow the various tools inter-operate in the back-end. For example, programmers and IT operations technicians can use textual syntax that resembles a programming language, while business product modellers 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 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

[7] See also Sect. 3.10.



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.*

probably be textual Kelly and Pohjonen, 2009. The textual syntax is also the cheapest to implement, especially with language workbenches like Xtext, Monticore, or Spoofax.

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 should 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. We discuss the design of graphical DSLs in **??**.

Use familiar, friendly, and intuitive notations, optimized for comprehension. It is well known that engineers are attracted to learning new languages, and that they tend to learn fast. This 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 (Karsai et al., 2009). (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

(Karsai et al., 2009; Wile, 2004). 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 appears bad from programming language perspective; he recalls a case of a DSL where parentheses were not balanced.

For the same reason choose known symbols for the known concepts. A plus ('+') should still mean addition and unlikely anything else, etc. If in need for 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* (Karsai et al., 2009). 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.

Exploit the examples and the meta-model to structure grammar productions. The least clear part of the 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 out 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 are enclosed with fixed opening and a 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

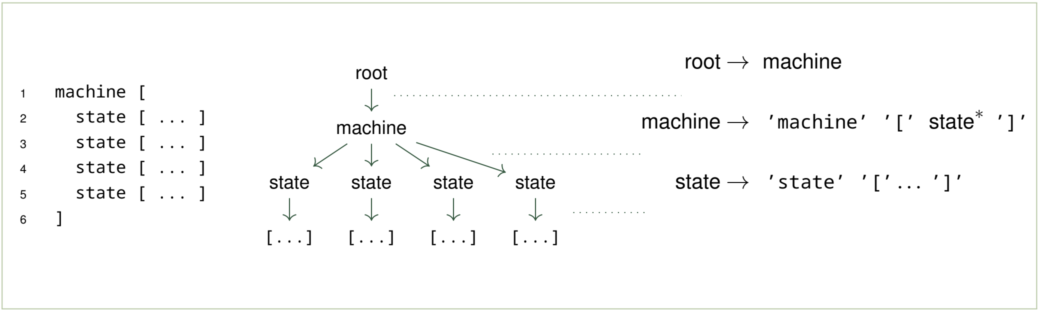


Figure 4.11: Picking up production nesting by parenthetical constructs. Left: the core nesting structure in Fig. 4.5, Center: a hypothetical core structure of a parse tree for the same example, Right: A hypothetical CFG able to generate the tree in the middle

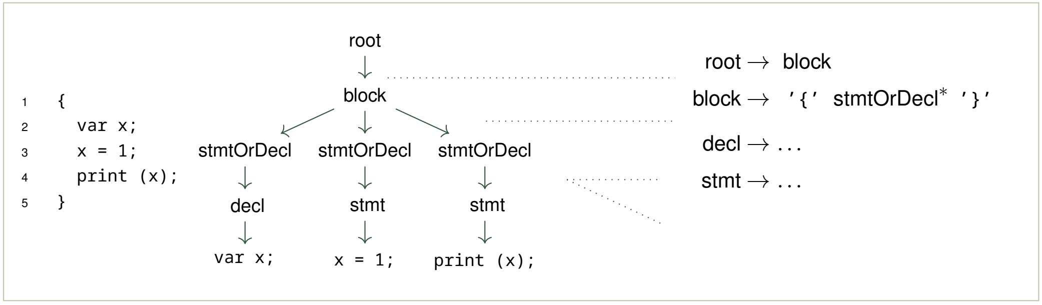


Figure 4.12: Creating an abstract nonterminal (stmtOrDecl) for elements appearing at the same level in a nesting structure of productions. The example uses a hypothetical Javascript-like syntax

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 advise 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 machine/ state dependency in Fig. 4.11. In order to do this, we introduce a new abstract nonterminal (called stmtOrDecl here) that admits both kinds of expansions, or use the alternative operator, for example:

$$\mathsf{block} \rightarrow \texttt{'\{'}\ (\mathsf{stmt} \mid \mathsf{decl})^{*}\ \texttt{'\}'}\ . \tag{4.27}$$

Finally, the semantic meaning of the parsed text often gives hints to creation of single rules: a convex 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. This is also seen in Fig. 4.10, where we chose a single non-terminal aExpr to group arithmetic expressions. A common top-level expression non-terminal will allow all kinds of expressions, wherever an expression is needed in the `robot` language, giving the language a nice design orthogonality.

> **Exercise 4.13.** Consider the subset of Cascading Style Sheets (CSS) studied in Exercise 3.2 and in Fig. 3.2, page 58. 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 could 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 subtree, but identify this subtree elsewhere, and link to it (reference it.) We have observed this mechanism in Line 25 of Fig. 4.6. Verify that the

[8]This is (roughly!) the procedure that the Xtext tool uses to generate an initial grammar for any given meta-model.

target property of a transition object in Fig. 3.1 is indeed a non-containment 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 (the left-biased recursive-descent parsers will not terminate on it). Indeed, a grammar which simply captures abstract syntax will lack a few details 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 by purely studying the meta-model in Fig. 3.21 on p. 87.

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 natural tool to mold the input data into an incompatible output structure. Parsing is difficult enough without this. It is better to try to move the abstract and concrete syntax close to each other. If you are experiencing an input-output impedance, design a new abstract-syntax meta-model structurally similar to the input format. Populate this new abstract-syntax during parsing (Karsai et al., 2009), 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 easier done in a general programming language, after parsing.

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 **??** for more about transformations).

Modularize your grammars (Alves and Visser, 2008). Grammars for real languages can get large. Modularize your grammar *vertically* and *horizontally*, not only to help reuse in other language project, 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 the 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.

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 system, 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 an additional advantage that it allows to limit handling white space to the low-level rules (see below, p. 127).

Many grammar specification languages allow importing parts of AST definitions and grammar fragments. For instance, Xtext allows importing 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. In the above, we have imported less—only the definitions of standard terminals, see 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.

Reuse existing grammars or parts of grammars. As mentioned above, once 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 set up. (Make sure you respect the licensing when you do so!)

Handle white space at the lexer level. In most situation, 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 tighter this would be needed (see below).

The situation gets more complex when using a parser combinator library. PEG are typically defined at the character level, like our example with

[9] https://www.antlr.org/

[10] http://www.metaborg.org/en/latest/

[11] http://www.txl.ca.

[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/

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 issues have been confined to the part shown in Fig. 4.8. The high-level productions in Fig. 4.7 have remained purely at the syntactic level, disregarding individual character issues. Ford (2004) 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 (what many other languages solves with braces), and Scala allows using line breaks as statement separators (what most C-family languages, including Java, do with semicolons). White-space-sensitive parsing, is a tempting design technique for DSLs as well: It allows to create models that are more concise, and may resemble human-aimed notation. For instance, in the Clafer Bak et al., 2016, we chose to use indentation and newlines to group and separate model elements, so that the models look more similarly to notes made by someone collecting main bullet points about a domain. White-space-sensitive syntaxes have also some disadvantages; chiefly it is hard to move pieces of code between places at different level in the blocks. They may also be confusing to programmers who trained with classical block-oriented syntax.

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 new-line character as a separator in the productions listing statements, declarations, etc. The 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 to experiment and to quickly hide defunct or underdeveloped parts of the model. They are an important usability feature (Karsai et al., 2009).

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 (2004) 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).

$$\textsf{comment} \to \ \texttt{'/*'}\,(\textsf{comment} \ | \ (!\ \texttt{'*/'}))^{*}\ \texttt{'*/'} \tag{4.28}$$

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 the, so called, until tokens, negative tokens, and hidden tokens[13] that help 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 from 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.

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 are growing and becoming increasingly complex, but some cases remain uncovered, and new bugs pop up all the time. We would like you to develop an intuition, 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 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 parentheses 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

[13] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#syntax

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 linguistic under the name the *pumping lemma for regular languages*. (We recommend the book by Hopcroft, Motwani, and Ullman (2001) for a thorough study of this theory.) 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.[14] 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.

$$\begin{array}{ll} \mathsf{poly} \rightarrow_1 \mathsf{poly}\ \mathsf{sign}\ \mathsf{var}\ \texttt{'\^{}'}\ \mathsf{num} \mid \varepsilon & \mathsf{var} \rightarrow_3 \texttt{'x'} \mid \texttt{'y'} \\ \mathsf{sign} \rightarrow_2 \texttt{'+'} \mid \texttt{'-'} & \mathsf{num} \rightarrow_4 \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \end{array} \tag{4.29}$$

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 1–2 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 "<<" could 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 are we 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.

Interestingly, the C++ grammar, prior to version C++11, was defined with a separate parsing and lexing phase, instantiating a list of lists of strings could not be written as above. Instead one should have written

[14] Technically a push-down automaton would suffice, but typical language definition tools give you a choice between regular expressions and grammars.

"`list<list<string> >`" separating the angle brackets, which is confusing for the users. The newer versions of C++ and Java, which uses a similar syntax for generics apparently do not suffer from the same problem.

Even if you need to tokenize based on the syntactic context, we recommend to limit this practice to the absolute minimum, and still perform most of tokenizing and parsing separately.

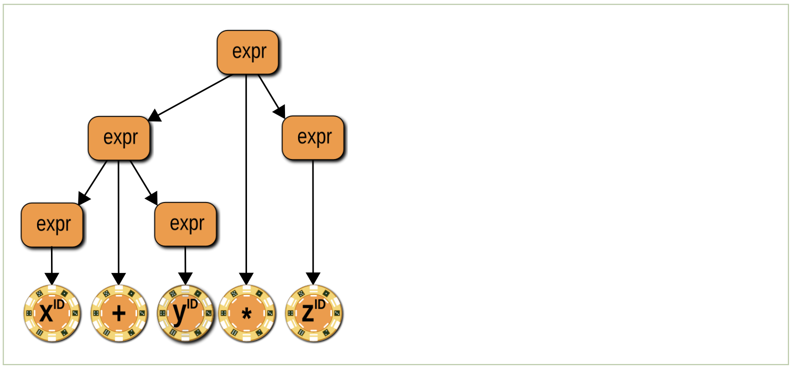


Figure 4.13: The left-most derivation tree for the expression $x+y*z$ using the grammar from Eqs. (4.14) to (4.16). Compare to Fig. 4.3

Avoid ambiguity in grammars. The grammar expression grammar from the beginning of the chapter (4.14–4.16) is ambiguous. The rightmost derivation shown therein (4.30) gives rise to the parse tree shown in Fig. 4.3. The following derivation, which is also right-most but picks the production rules in a different order, leads to the tree in Fig. 4.13. A different tree! Check!

$$\begin{aligned}
\text{expr} &\rightarrow_2 \text{expr}\ \texttt{'*'}\ \text{expr}\\
&\rightarrow_3 \text{expr}\ \texttt{'*'}\ \texttt{ID}\\
&\rightarrow_1 \text{expr}\ \texttt{'+'}\ \text{expr}\ \texttt{'*'}\ \texttt{ID}\\
&\rightarrow_3 \text{expr}\ \texttt{'+'}\ \texttt{ID}\ \texttt{'*'}\ \texttt{ID}\\
&\rightarrow_3 \texttt{ID}\ \texttt{'+'}\ \texttt{ID}\ \texttt{'*'}\ \texttt{ID}
\end{aligned}\tag{4.30}$$

Exercise 4.16. Write down a *left-most* 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 left-most 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 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 the ambiguity problems. Most parsing tools allow to specify 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 the 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 the easiest to start with 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 (Cordy, 2006) 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 the 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, the 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. Formally, we define left recursion as follows:

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 $n \to^{+} n\alpha$ for some arbitrary string of symbols α. (Aho et al., 2006)*

In other words, the 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 left-most symbol again. Productions (4.16) and (4.15) are both left-recursive. You are encouraged to convince yourself that none of the productions in Figs. 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, Xtext however does not make 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 restriction of Parboiled2, ANTLR, Xtext, and of 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 it, 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 (like Xtext or Parboiled2).

Let us simplify our expression grammar for a moment, to just 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:

$((w+x)+y)+z$	the left-associative,
$(w+x)+(y+z)$	a balanced one,
$w+(x+(y+z))$	the right-associative.

If you cannot see why these trees arise, try to write out the corresponding derivations. To eliminate left recursion, we would like to disallow arbitrary parse trees, and focus the parser on one particular format, the last one listed above. The parsing $w+(x+(y+z))$ makes it particularly clear that we can see a complex arithmetic summation just as a sequence of additions, which always starts with an identifier and then it is followed by more identifiers, separated by addition symbols. This really looks like a plus-separated list of identifiers! A standard grammar generating a comma-separated list of identifiers is not left-recursive (Try to write it out! see Exercise 4.34). We should be able to model a list of additions the same way.

If you see it like that, there is no inherent left-recursion in parsing long summations. The left-most symbol in an input string is always a known terminal, here an `ID`, not a full-blown expression. This suggest the following grammar transformation:

$$\text{expr} \to \ \texttt{ID}\ (\ \texttt{'+'}\ \texttt{ID})^*\ . \tag{4.34}$$

After this change there is no more left recursion left, 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.

Let us consider the original example again, where we had two interdependent 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:

$$\textsf{expr} \to \ \textsf{term} \ (\ \texttt{'+'} \ \textsf{term})^* \ . \tag{4.36}$$

$$\textsf{term} \to \ \textsf{term} \ \texttt{'*'} \ \textsf{term} \ | \ \texttt{ID} \tag{4.37}$$

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:

$$\textsf{expr} \to \ \textsf{term} \ (\ \texttt{'+'} \ \textsf{term})^* \tag{4.38}$$

$$\textsf{term} \to \ \texttt{ID} \ (\ \texttt{'*'} \ \texttt{ID})^* \ . \tag{4.39}$$

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. They key to get there, was to apply the rewrite from Fig. 4.14 twice, once per each case, and also to observe that different precedence expression should be represented by different non-terminals (stratified), where we can use Kleene iteration at each level. The following figure summarize the general rule for grammars with left recursion in multiple cases:

$$\boxed{n \to \beta \mid n\alpha n \mid n\gamma n} \quad \rightsquigarrow \quad \boxed{\begin{array}{c} n \to m \ (\alpha m)^* \\ m \to \beta \ (\gamma\beta)^* \end{array}}$$

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 is 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:

$$\textsf{expr} \to \ \texttt{ID} \ | \ \texttt{'('} \ \textsf{expr} \ \texttt{')'} \ | \ \textsf{expr} \ \texttt{'+'} \ \textsf{expr} \ | \ \textsf{expr} \ \texttt{'*'} \ \textsf{expr} \tag{4.40}$$

and the unambiguous non-left recursive grammar, where precedence has been enforced. (A factor is an expression that can be multiplied.)

$$\begin{aligned}\text{expr} &\rightarrow \text{term}\ (\ \texttt{'+'}\ \text{term}\ )^{*}\\ \text{term} &\rightarrow \text{factor}\ (\ \texttt{'*'}\ \texttt{factor})^{*}\\ \text{factor} &\rightarrow \texttt{ID}\ \mid\ \texttt{'('}\ \text{expr}\ \texttt{')'}\end{aligned}\tag{4.41}$$

Convince yourself that both 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 writing a grammar for a complex language in a single seating. 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 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 this 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. Optimization tends to eliminate excessive non-terminals and rules (Alves and Visser, 2008). Optimization might give you a faster parser, but most importantly it helps you to understand your parser well. It helps 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

```
"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/mdsebook/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.*

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 Scala to make writing Xtext tests in Scala more idiomatic.[15] 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 the *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.

[15] source: xtext.scala/src/main/scala/mdsebook/xtext/scala/XtextScala.scala

```
"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/mdsebook/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.*

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 grammar directly. This is why our Xtext tests invoked the top-level `Model` rule. An independent project provides facilities for production-level tests.[16]

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 concious of white space when reading—it is only important if its absence would cause a confusion. For instance, in 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 if the elements required to be optional can be omitted, and if 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 for your operators. This is always the case if your expressions have side-effects. For operator precedence, compare ASTs both with and without parentheses, to check if 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 that 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.

[16] https://github.com/itemis/xtext-testing

- *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 input strings 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 shall 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 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 (Bentley, 1986). In this chapter, the model in Fig. 4.5 was created to fulfil this role. The key advantage of this tactics is that it can be implemented very fast.

Exercise 4.17. Consider the following simple grammar for a subset of Cascading Style Sheets.[a] 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.

$$\begin{aligned}
\mathsf{css} &\to \mathsf{specification}^* \\
\mathsf{specification} &\to \mathsf{element}\ \texttt{'\{'}\ \mathsf{attribute}^*\ \texttt{'\}'} \\
\mathsf{element} &\to \texttt{'p'} \mid \texttt{'div'} \\
\mathsf{attribute} &\to \mathsf{attrID}\ \texttt{':'}\ \texttt{color}\ \texttt{';'} \\
\mathsf{color} &\to \texttt{'black'} \mid \texttt{'white'} \mid \texttt{'red'} \\
\mathsf{attrID} &\to \texttt{'color'} \mid \texttt{'background-color'} \qquad (4.42)
\end{aligned}$$

4.7 Meta-hierarchy for Grammars

Let us step back for a moment and look at Figs. 4.6 and 4.7 again. Both figures present language definitions. They define what models can be

[a] https://www.w3.org/Style/CSS/Overview.en.html

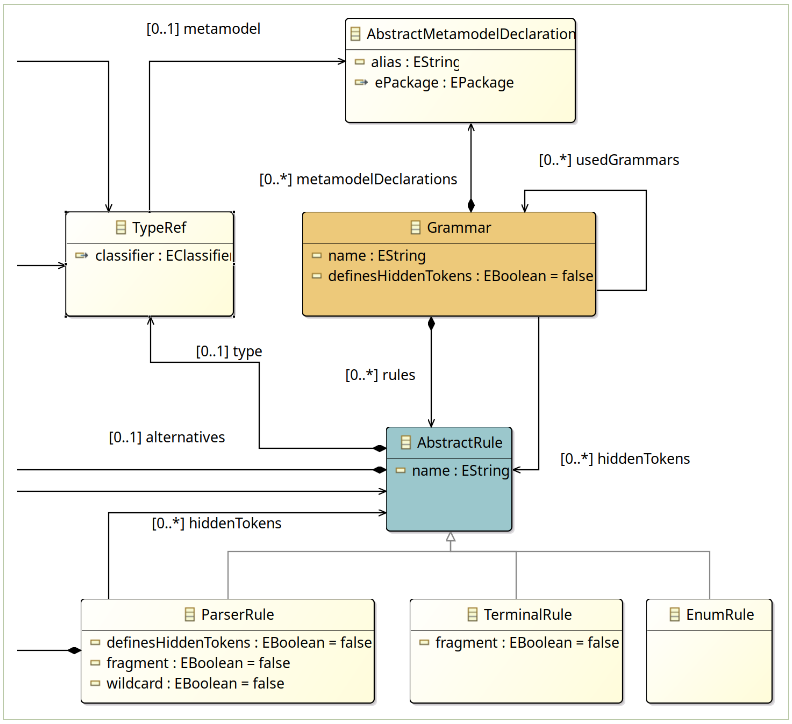


Figure 4.18: A fragment of the meta-model used in the implementation of the Xtext framework for representing grammars.

written in the finite state machines 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 for them (for instance using meta-modeling) and we can design grammars for this language (using grammars!). Figure 4.18 presents a fragment of the original 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 Table 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

[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 January 2020). Our code repository provides a laid out diagram, which was used to create the Fig. 4.18. See figures/model/Xtext.aird

practice has spread from the compiler community to the broader community building language infrastructures in general. Thus Ecore models are represented as instances of 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, 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 less powerful papier-mâché version of the tool. The early prototype shall powerful enough to build the first real boot-strapped 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[a]). 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, 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 controlling 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:

[a] https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)

```
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).*

$$\begin{aligned}
\textsf{regex} &\rightarrow \textsf{regex}\ \texttt{'|'}\ \textsf{regex} & \textsf{regex} &\rightarrow \texttt{'('}\ \textsf{regex}\ \texttt{')'} \\
\textsf{regex} &\rightarrow \textsf{regex}\ \textsf{regex} & \textsf{regex} &\rightarrow \texttt{'a'} \quad \text{for any character } \texttt{a} \in \Sigma \\
\textsf{regex} &\rightarrow \textsf{regex}\ \texttt{'+'} & \textsf{regex} &\rightarrow \varepsilon
\end{aligned} \tag{4.43}$$

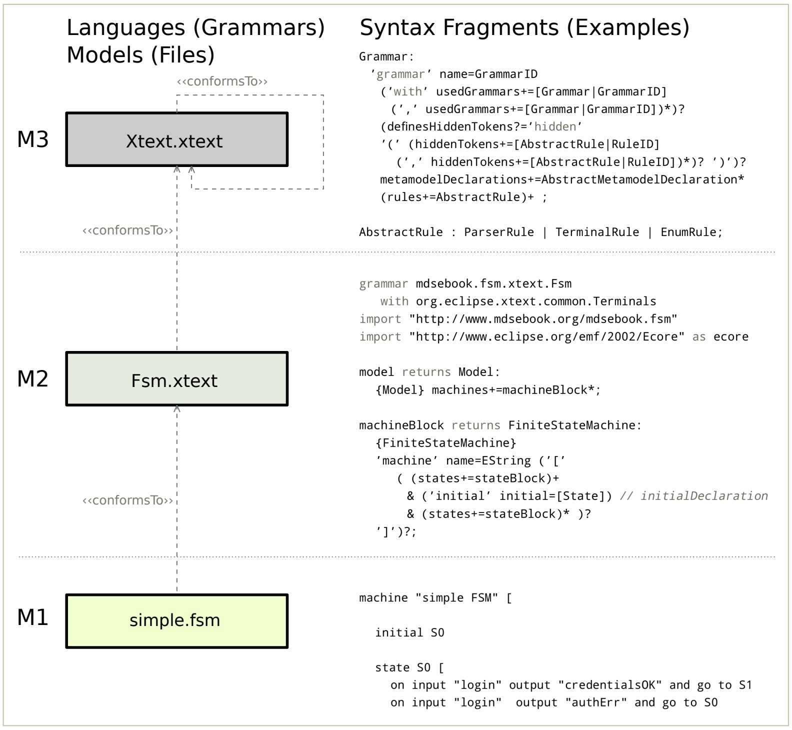
Incidentally, the above grammar is ambiguous and left-recursive. Exercise 4.44 considers transforming it into a non-left-recursive form. Importantly, the pipe symbol above is a terminal symbol, not the alternative operator from EBNF (this is why we quoted it). Similarly the quoted plus symbol, and the quoted parentheses are not EBNF parentheses or Kleene iteration from EBNF. When writing grammars for a grammar DSL, we face the same cognitive confusions we had seen 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 simplest context-free grammars, following Def. 4.18. Warning: Do not be surprised—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 Table 4.2. Compare your grammar to the official Xtext grammar.[18] 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

[18] 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;
```

```
grammar mdsebook.fsm.xtext.Fsm
   with org.eclipse.xtext.common.Terminals
import "http://www.mdsebook.org/mdsebook.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: *Hierarchy of concrete syntax languages (with the finite state machine language in the bottom). Compare to Fig. 3.13*

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. Because of the 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 so called *Dragon Book* by Aho et al. (2006). However, many competing books exist and most of them are very good. A more recent concise reference has been authored by Mogensen (2011). 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 warmly encouraged to delve deeper into the subject, starting with the above two volumes.

The documentation of Parboiled2[19] is helpful if you need to learn using the combinators. Myltsev (2019) describes the design principles, and the implementation of the Parboiled2 tool. Chiusano and Bjarnason (2014) devote an entire chapter to the case study of design of a parser combinator library in Scala (see Chapter 9). Interestingly, as of today, the problem whether PEGs and CFGs are incomparable is still open. Ford (2004) has shown that there exist 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 (2018) shown 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 (1965) was probably the first to propose a conservative procedure for deciding the problem, based on detecting the LR(k) shift–reduce conflicts. More recently, Brabrand, Giegerich, and Møller (2010) give a short account of 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, the GLL parsing methods (Lang, 1974) are entering mainstream tools extremely slowly. Open source libraries are only starting to appear and gather initial interest.[20] 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. (2006). Medeiros, Mascarenhas, and Ierusalimschy (2012) 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 provides extensive documentation[21] for language designers. We have included a condensed summary in Appendix C. Bettini (2013) has written a book on the framework, the best reading material on the topic so far.

Additional Exercises

Exercise 4.22. Explain in English what are the languages described by the following regular expressions; Parentheses are meta-operators used for grouping, and ␣ denotes a single blank character:

a) $(10)^*$, **b)** $1(0|1)^*$, **c)** $(\ (0(1|2)3)␣)^+$.

Examples: The 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.

[19] Available at their GitHub page https://github.com/sirthias/parboiled2, seen January 2020

[20] Examples: https://github.com/rust-lang/gll for Rust, https://github.com/djspiewak/gll-combinators for Scala, seen January 2020

[21] http://www.eclipse.org/Xtext/, seen January 2020

Exercise 4.23. Decide if 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.

$$L = \{b, aab, aaaab, aaaaaab, \dots\} \cup \{c, ac, aac, aaac, aaaac, \dots\} \qquad (4.44)$$

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 3 digits precision. There are no restrictions on the leftmost and the right-most 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 regular 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 would be the only symbol before or after the point.

Exercise 4.30. Write a regular expression matching a correct cardinality expression of the Clafer language (Bak et al., 2016), 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:

$$\begin{array}{llll} \mathtt{s} \to_1 \mathtt{t\ u\ 'a'\ v} & & \mathtt{u} \to_3 \mathtt{'reads'} \mid \mathtt{'writes'} \\ \mathtt{t} \to_2 \mathtt{'John'} \mid \mathtt{'Mary'} \mid \mathtt{'Alice'} & & \mathtt{v} \to_4 \mathtt{'book'} \mid \mathtt{'letter'} \mid \mathtt{'poem'} \end{array}$$

Exercise 4.33. Explain in English what language is generated by the following EBNF grammar, with `s` being the start symbol.

$$\mathtt{s} \to_1 \mathtt{s\ op\ id} \mid \mathtt{id} \qquad \mathtt{op} \to_2 \mathtt{'->'} \mid \mathtt{'.'} \qquad \mathtt{id} \to_3 \mathtt{'x'}$$

Exercise 4.34. Consider the following context-free grammar for a comma-separated list of identifiers, where the start symbol is `s` and the terminal `ID` refers to a standard Java identifier token.

$$\mathtt{s} \to_1 \mathtt{'('\ t\ ')'} \qquad \mathtt{t} \to_2 \mathtt{ID\ ','\ t} \qquad \mathtt{t} \to_3 \varepsilon \tag{4.45}$$

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 accepted by the grammar above.

Exercise 4.35. Specify concrete syntax for a comma-separated list of hexadecimal numbers. Each number is built of arbitrary number of white-space-separated groups of digits. Each group of digits consists of four digits, except for the leftmost (most-significant) group which can contain less. Decide whether to use regular expressions, grammars, or both to solve the task, and argue for your choice. *Positive example:* `'c0 ffee, ff, f10 abcd 0123'`. *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.

$$\mathtt{s} \to \mathtt{a\ b\ c} \qquad \mathtt{b} \to \mathtt{b\ '1'} \mid \varepsilon \qquad \mathtt{a} \to \mathtt{a\ '2'} \mid \varepsilon \qquad \mathtt{c} \to \mathtt{c\ '3'} \mid \varepsilon$$

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. In the following grammar, s is the start symbol. Show *two different* derivations of *different length* of two different strings from this grammar. Mark the derivation arrows with production numbers, so that it is easy to reconstruct the order in which the rules are applied.

$$
\begin{array}{lll}
\texttt{s} \rightarrow_1 \texttt{'a' 'b' s} & \texttt{s} \rightarrow_3 \texttt{'(' s ')'} & \texttt{s} \rightarrow_5 \texttt{'d'} \\
\texttt{s} \rightarrow_2 \texttt{'g'} & \texttt{s} \rightarrow_4 \texttt{'a' 'b' s} &
\end{array}
\tag{4.46}
$$

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 logics 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. 142.

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, so that you can example. For instance, transcribe "MDSE IS FUN". Write an EBNF grammar defining a message in the Morse code over the set of the above three 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 2–3 more variants of this example. For instance, consider whether it is required to use let definitions, or if the definitions can be nested directly in the "simpleFSM," can naming of finite state machines be optional? Then write an abstract EBNF grammar generating this language or use your favourite syntax specification tool to implement it.

Exercise 4.47. Parsing cardinality expressions, like those in Exercise 4.30, using a regular expression is suboptimal. Separating elements of the expression is clumsy, and it is somewhat messy to control the white-space. Write an EBNF grammar for Clafer's cardinality expression, following a slightly reacher specification than the one 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 page 87 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, the program expression grammars are deterministically processed from left to right, and once a rule matches, a typical PEG parser is not backtracking. 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:

$$\text{inputClause} \rightarrow \texttt{'on' 'input' ID} \mid \texttt{'on' ID} \tag{4.48}$$

Does reordering (swapping) the two operands of the alternative in the above production affects the language this production accepts? *Reflection points:* Ford (2004) 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 (2004), 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.[22]

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 train the use of predicates in PEGs. If you seek an example, Ford (2004) 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.

[22]More about predicates in Parboiled2 grammars at https://github.com/sirthias/parboiled2.

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 loosing 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 page 185. See also Exercise 3.34 on page 91.

References

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools. Edition 2*. Prentice Hall.

Alves, Tiago L. and Joost Visser (2008). "A Case Study in Grammar Engineering". In: *SLE*. Vol. 5452. Lecture Notes in Computer Science. Springer, pp. 285–304.

Bak, Kacper et al. (2016). "Clafer: unifying class and feature modeling". In: *Software and System Modeling* 15.3, pp. 811–845. DOI: 10.1007/s10270-014-0441-1. URL: http://dx.doi.org/10.1007/s10270-014-0441-1.

Bentley, Jon (Aug. 1986). "Programming Pearls: Little Languages". In: *Commun. ACM* 29.8, pp. 711–721. ISSN: 0001-0782. DOI: 10.1145/6424.315691. URL: http://doi.acm.org/10.1145/6424.315691.

Bettini, Lorenzo (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt.

Brabrand, Claus, Robert Giegerich, and Anders Møller (2010). "Analyzing ambiguity of context-free grammars". In: *Sci. Comput. Program.* 75.3, pp. 176–191. DOI: 10.1016/j.scico.2009.11.002. URL: https://doi.org/10.1016/j.scico.2009.11.002.

Chiusano, Paul and Rúnar Bjarnason (2014). *Functional Programming in Scala*. Manning.

Chomsky, Noam (1957). *Syntactic Structures*. Mouton & Co.

Cordy, James R. (2006). "The TXL source transformation language". In: *Sci. Comput. Program.* 61.3, pp. 190–210. DOI: 10.1016/j.scico.2006.04.002. URL: https://doi.org/10.1016/j.scico.2006.04.002.

Ford, Bryan (2004). "Parsing expression grammars: a recognition-based syntactic foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, pp. 111–122. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964011. URL: https://doi.org/10.1145/964001.964011.

Frost, R. and John Launchbury (1989). "Constructing Natural Language Interpreters in a Lazy Functional Language". In: *Comput. J.* 32.2, pp. 108–121. DOI: 10.1093/comjnl/32.2.108. URL: https://doi.org/10.1093/comjnl/32.2.108.

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

Karsai, Gabor et al. (2009). "Design Guidelines for Domain Specific Languages". In: *9th OOPSLA Workshop on Domain-Specific Modeling*. URL: http://www.dsmforum.org/events/DSM09/Papers/Karsai.pdf.

Kelly, Steven and Risto Pohjonen (2009). “Worst Practices for Domain-Specific Modeling”. In: *IEEE Software* 26.4, pp. 22–29.

Klint, Paul, Ralf Lämmel, and Chris Verhoef (2005). “Toward an engineering discipline for grammarware”. In: *ACM Trans. Softw. Eng. Methodol.* 14.3, pp. 331–380. DOI: 10.1145/1072997.1073000. URL: https://doi.org/10.1145/1072997.1073000.

Knuth, Donald E. (1965). “On the Translation of Languages from Left to Right”. In: *Information and Control* 8.6, pp. 607–639. DOI: 10.1016/S0019-9958(65)90426-2. URL: https://doi.org/10.1016/S0019-9958(65)90426-2.

Lang, Bernard (1974). “Deterministic Techniques for Efficient Non-Deterministic Parsers”. In: *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken, Germany, July 29 - August 2, 1974, Proceedings*. Ed. by Jacques Loeckx. Vol. 14. Lecture Notes in Computer Science. Springer, pp. 255–269. ISBN: 3-540-06841-4. DOI: 10.1007/3-540-06841-4_65. URL: https://doi.org/10.1007/3-540-06841-4%5C_65.

Loff, Bruno, Nelma Moreira, and Rogério Reis (2018). “The Computational Power of Parsing Expression Grammars”. In: *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*. Ed. by Mizuho Hoshi and Shinnosuke Seki. Vol. 11088. Lecture Notes in Computer Science. Springer, pp. 491–502. ISBN: 978-3-319-98653-1. DOI: 10.1007/978-3-319-98654-8_40. URL: https://doi.org/10.1007/978-3-319-98654-8%5C_40.

Medeiros, Sérgio, Fabio Mascarenhas, and Roberto Ierusalimschy (2012). “Left Recursion in Parsing Expression Grammars”. In: *CoRR* abs/1207.0443. arXiv: 1207.0443. URL: http://arxiv.org/abs/1207.0443.

Mogensen, Torben Ægidius (2011). *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, pp. I–XXI, 1–204. ISBN: 978-0-85729-828-7, 978-0-85729-829-4.

Myltsev, Alexander A. (2019). “parboiled2: a macro-based approach for effective generators of parsing expressions grammars in Scala”. In: *CoRR* abs/1907.03436. arXiv: 1907.03436. URL: http://arxiv.org/abs/1907.03436.

Wile, David S. (2004). “Lessons learned from real DSL experiments”. In: *Sci. Comput. Program.* 51.3, pp. 265–290.

Plenitude, when too plenitudinous,
was worse than destitution,
for—obviously—what could one do,
if there was nothing one could not?
(Lem, 1967)

5 Static Semantics

5.1 Why Static Semantics?

In Chapter 3 (see also Appendix A), 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.

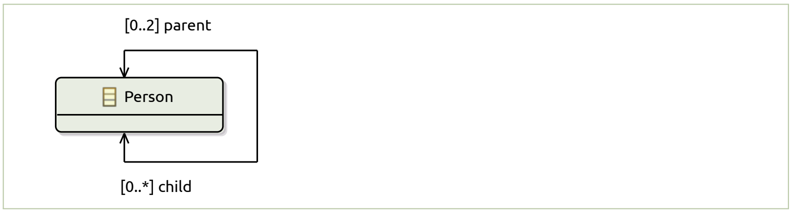


Figure 5.1: An Ecore class diagram with two unidirectional references (overlayed on top of each other) for the `Person` *class*

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.

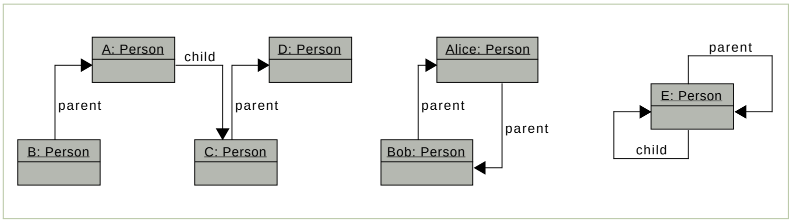


Figure 5.2: Three undesirable instances admitted by the diagram of Fig. 5.1. The diagrams show complete models, not partial views.

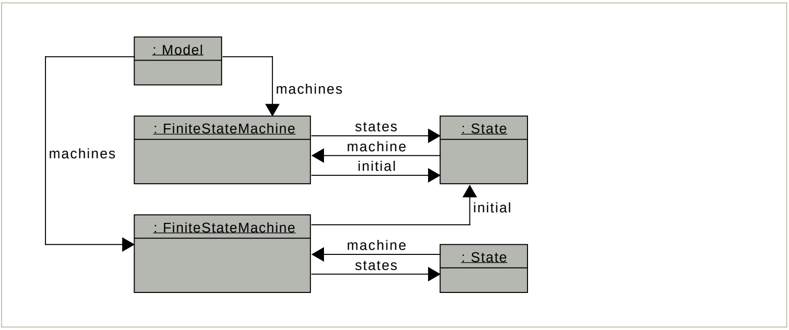


Figure 5.3: *An unexpected instance for the FSM meta-model of Fig. 3.1, with one machine 'borrowing' its initial state from another one*

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 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 finite state machine meta-model (Fig. 3.1, page 58), we would like to require that the initial state of each machine is also its own state. An instance violating this requirement is presented 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 referencing a state from another state machine, at the cost of creating a new problem: the collection of `states` would no longer contain all states, as each 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 1–5 define a class `Person` with collections `parent` and `child`. This class can only be

[1] In Ecore, one can enforce that two references are opposites by setting the `eOpposite` property (http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/EReference.html#getEOpposite())

```
// 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.*

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 not using 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 main idea is found in lines 1–3, where `parent` and `child` are properties that resolve to a lazy list of persons with a *delay*. Because the value of parent and child are placed in a function value, a lambda, 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. The remaining part of the code provides a more convenient API to construct the objects. If you are not a Scala programmer, feel free to skip to the next paragraph. Lines 6–10 define a convenience constructor (a factory function) that uses call-by-name to create Persons without evaluating the child/parent objects. This allows constructing objects without explicit

```scala
class Person ( name: String,
               parent: ()=>LazyList[Person],
               child: ()=>LazyList[Person] )

object Person {
  def apply (
    name: String,
    parent: =>LazyList[Person] = empty,
    child: =>LazyList[Person] = empty): Person =
      new Person (name, ()=>parent, ()=>child)
}
```
source: person.scala/src/main/scala/Person.scala

Figure 5.5: *A Scala ADT for representing persons purely; corresponds to the Ecore model in Fig. 5.1. This design can be instantiated, and can represent the duality of parent–child relationships, unlike Fig. 5.4, but cannot prevent cycles.*

```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/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.*

lambdas, bringing the experience closer to a lazy programming language like Haskell.[2] 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 opposite of each other 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 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

[2]The convenience constructor is needed, because class constructors in Scala can only have by-value arguments. In a lazy programming language, like Haskell, Fig. 5.5 could be reduced to lines 1–3.

more intricate restrictions than modeling languages and type systems are able to express. We shall best 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 page 54. 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 formulating 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

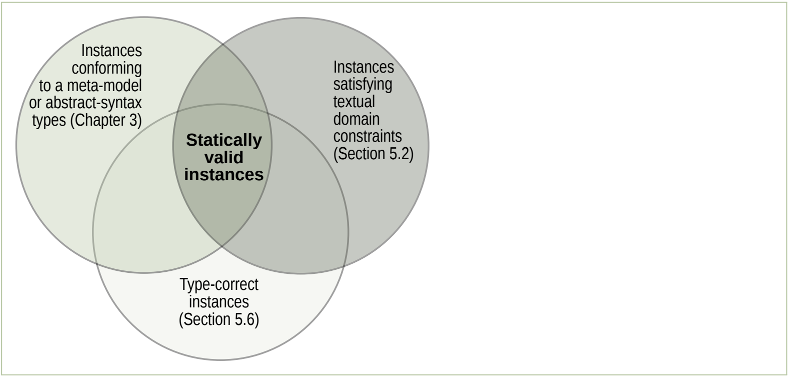

Instances conforming to a meta-model or abstract-syntax types (Chapter 3)
Instances satisfying textual domain constraints (Section 5.2)
Statically valid instances
Type-correct instances (Section 5.6)

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.*

cheaper and simpler method, suitable for small languages used commonly in model-driven development (Sections 5.2–5.5). In Sect. 6.8 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 to greatly simplify other components of a tool chain. An explicit definition of static semantics also leads to 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.

5.2 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 of 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 a 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 an intersection of the diagram and the constraints. Table 5.1 defines the core semantics of class diagrams by translation to first-order predicate logics. 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 Table 5.1 row-by-row, just enough to develop a logics-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

Table 5.1: *Mapping core concepts of class diagrams to first-order predicate logic. Notation: $\forall$ = for all (universal quantification), $\rightarrow$ = 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 $\forall x.\, \mathsf{C}(x) \rightarrow \mathsf{D}(x)$
Non-containment reference r from class C to D, $\mathsf{C.r : D}$	$\rightsquigarrow$	A binary predicate $\mathsf{r}(x,y)$ true if reference `r` from x points to y and a constraint $\forall x,y.\, \mathsf{r}(x,y) \rightarrow \mathsf{C}(x) \wedge \mathsf{D}(y)$
Containment reference from class C to D, $\mathsf{C.r : D}$	$\rightsquigarrow$	Same as the non-containment reference plus a constraint that $\forall x,y.\, \mathsf{r}(x,y) \rightarrow \mathsf{owns}(x,y)$, where $\mathsf{owns}(x,y)$ is a special predicated shared between all references in the diagram such that $\forall y.\ |\{x \mid \mathsf{owns}(x,y)\}| \leq 1$.
Cardinality constraint `[a..b]` on reference r in class C	$\rightsquigarrow$	A constraint $\forall x. \mathsf{C}(x) \rightarrow |\{y \mid \mathsf{r}(x,y)\}| \in [\mathsf{a};\mathsf{b}]$
Attribute a of type T in a class C, $\mathsf{C.a : T}$	$\rightsquigarrow$	Formally the same as non-containment reference: a binary predicate $\mathsf{a}(x,y)$ true if the value of a in x is y and a constraint $\forall x,y. \mathsf{a}(x,y) \rightarrow \mathsf{C}(x) \wedge \mathsf{T}(y)$, where $\mathsf{T}(y)$ holds iff T is the type of y
References r_1, r_2 are opposite	$\rightsquigarrow$	A constraint $\forall x,y.\, \mathsf{r}_1(x,y) \equiv \mathsf{r}_2(y,x)$

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 Table 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 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 Table 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:

$$\forall x.\, \forall y.\, \mathsf{states}(x,y) \to \mathsf{FiniteStateMachine}(x) \wedge \mathsf{State}(y) \tag{5.7}$$

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 Table 5.1).

$$\forall x.\, \forall y.\, \mathsf{machine}(x,y) \to \mathsf{State}(x) \wedge \mathsf{FiniteStateMachine}(y) \tag{5.8}$$

$$\forall x.\, \forall y.\, \mathsf{machine}(x,y) \equiv \mathsf{states}(y,x) \tag{5.9}$$

Moreover a machine is composed of states, so it owns them all, and this ownership cannot be shared with any other class, as indicated by the black diamond symbol 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:

$$\forall x.\,\forall y.\,\mathsf{states}(x,y) \to \mathsf{owns}(x,y) \tag{5.10}$$
$$\forall y.\ |\{x \mid \mathsf{owns}(x,y)\}| \leq 1 \tag{5.11}$$

A cardinality constraints 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 (cr. row 5 in Table 5.1):

$$\forall x.\,\mathsf{Machine}(x) \to |\{y \mid \mathsf{states}(x,y)\}| \geq 1 \tag{5.12}$$
$$\forall x.\,\mathsf{State}(x) \to |\{y \mid \mathsf{machine}(x,y)\}| = 1 \tag{5.13}$$

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. 102.

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 Table 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 out the characteristic first-order formula (the diagrammatic constraint) for the meta-model in Fig. 5.1. Then consider two rightmost instances in Fig. 5.2 and convince yourself that it satisfies 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 to formal logics? 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 rewritten to *"all objects that are **machines** must have distinct **names**,"* or to *"if an object is a **machine**, then it has 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.*"

$$\begin{aligned}&\forall m_1.\forall m_2.\forall n_1.\forall n_2.\ \underline{m_1 \neq m_2 \wedge}\\ &\qquad \underline{\mathsf{FiniteStateMachine}(m_1) \wedge \mathsf{FiniteStateMachine}(m_2) \wedge}\\ &\qquad \underline{\mathsf{name}(m_1,n_1) \wedge \mathsf{name}(m_2,n_2)} \quad \rightarrow \quad n_1 \neq n_2 \end{aligned} \tag{5.15}$$

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 from 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:

$$\begin{aligned}&\forall m.\forall s_1.\forall s_2.\forall n_1.\forall n_2.\\ &\quad \underline{s_1 \neq s_2 \wedge \mathsf{states}(m,s_1) \wedge \mathsf{states}(m,s_2)\ \wedge \mathsf{name}(s_1,n_1) \wedge \mathsf{name}(s_2,n_2)}\\ &\qquad \rightarrow \quad n_1 \neq n_2 \end{aligned} \tag{5.16}$$

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, both for **C1** and **C2**—names enforces the second argument to be a name. However, this predicate does not help restricting 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 types 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 logics, a quantifier binds a variable. A correct constraint in logics 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. 157: *"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:

$$\begin{aligned}\forall t.\forall s_1.\forall s_2.\forall m.\ &\mathsf{source}(t,s_1) \wedge \mathsf{target}(t,s_2) \wedge \mathsf{machine}(s_1,m)\\ &\rightarrow \mathsf{machine}(s_2,m)\end{aligned} \tag{5.19}$$

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:

$$\begin{aligned}
&\forall M.\forall m.\forall t.\\
&\qquad(\mathsf{machines}(M,m) \wedge \exists s_1.\ \mathsf{states}(m,s_1) \wedge \mathsf{leavingTransitions}(s_1,t))\\
&\qquad\quad \to (\forall s_2.\ \mathsf{target}(\mathsf{t},\mathsf{s_2}) \to \mathsf{machines}(m,s_2)) \qquad (5.20)
\end{aligned}$$

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 logics 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 so, the software-oriented specification languages do offer a lot of devices to make your life easier and the 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. 5.3.

> **Exercise 5.4.** Recall the determinism constraint **C6** from Example 15 on p. 157: *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 Table 5.1.

Even though the first-order logics 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, that commonly appear in modeling languages cannot be captured in first-order logics. 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.[3] 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 logics cannot express them. The problem is that connectedness properties are global properties of a graph, while in first-order logics 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 logics.[4] It requires a second-order logic, where we can constraint 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 the second-order logics:

$$\forall m.\forall s_1.\forall s_2.\quad \mathsf{initial}(m,s_1) \wedge \mathsf{states}(m,s_2) \quad\rightarrow\quad \mathsf{successor}^*(s_1,s_2)\ , \tag{5.21}$$

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 $\mathsf{successor}$.

In verbose English Eq. (5.21) says: *If s_1 is an **initial** state of **machine** m, and if s_2 is its other **state**, 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 s_2 state as the **target**.* Notice, that since $\mathsf{successor}$ 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 $\mathsf{successor}$ is merely a convenience here. On the other hand, the use of the *reflexive transitive closure* of a predicate, denoted by the asterisk in **C5** is key. Let us define it semi-formally first:

[3]The direction of a transition in this case is from its source to 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.

[4]For the interested reader, this follows from the compactness theorem for first-order logics

$$\begin{aligned}\mathsf{successor}^*(s_1,s_n) \quad &\equiv \quad (s_1 = s_n) \ \vee \\ (\exists s_2 \cdots \exists s_{n-1}.\, \mathsf{successor}(s_1,s_2) \wedge &\mathsf{successor}(s_2,s_3) \wedge \cdots \wedge \mathsf{successor}(s_{n-1},s_n))\end{aligned} \tag{5.22}$$

A reflexive transitive closure of a binary predicate is a new binary predicate that is reflexive—so that $\mathsf{sucessor}^*(s,s)$ 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, ..., s_{n-1}. The intermediate states and even their number n are different for each pair of arguments. This is in stark contrast with a fixed arity of predicates in first-order logics. Equation (5.22) cannot be written in the first order logics without the informal dots in the quantification.

More precisely, we define the reflexive transitive closure operator as the smallest predicate $\mathsf{successor}^*$ 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 for 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 the first-order logics.

Exercise 5.5. Prove that the predicate $\mathsf{successor}^*(s_1,s_n) \equiv \mathsf{true}$ 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 logics, 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.

5.3 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

C1. *All machines must have distinct names.*

$$\forall m_1.\forall m_2.\forall n_1.\forall n_2.\ m_1 \neq m_2 \land \mathsf{FiniteStateMachine}(m_1) \land \mathsf{FiniteStateMachine}(m_2) \land \mathsf{name}(m_1,n_1) \land \mathsf{name}(m_2,n_2) \rightarrow n_1 \neq n_2$$

```
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.*

$$\forall m.\forall s_1.\forall s_2.\forall n_1.\forall n_2.\ s_1 \neq s_2 \land \mathsf{states}(m,s_1) \land \mathsf{states}(m,s_2) \land \mathsf{name}(s_1,n_1) \land \mathsf{name}(s_2,n_2) \rightarrow n_1 \neq n_2$$

```
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).*

$$\forall t.\forall s_1.\forall s_2.\forall m.\ \mathsf{source}(t,s_1) \land \mathsf{target}(t,s_2) \land \mathsf{machine}(s_1,m) \rightarrow \mathsf{machine}(s_2,m)$$

```
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.*

$$\forall m.\forall s_1.\forall s_2.\ \mathsf{initial}(m,s_1) \land \mathsf{states}(m,s_2) \rightarrow \mathsf{successor}^*(s_1,s_2)$$

where

$$\mathsf{successor}^*(s_1,s_n) \equiv (s_1 = s_n) \lor \exists s_2.\ \mathsf{successor}(s_1,s_2) \land \mathsf{successor}^*(s_2,s_n)$$

```
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) seen1
  else BFS (toSee.flatMap {succ _}.diff (seen1), seen1)
}
def succ (s: State): Seq[State] =
  s.getLeavingTransitions
   .asScala.map { _.getTarget }.toSeq
```

The transitive closure operation is implemented as a custom recursive algorithm (`reachable`), but not that the main constraint (line 1), which is still kept in a simple declarative sentential form.

source: fsm.scala/src/main/scala/mdsebook/fsm/scala/Constraints.scala

Table 5.2: *Mathematical logics (left) vs implementations of constraints in a programming language (Scala, right). We use our Scala-Ecore integration layer scala/src/main/scala/mdsebook/scala/EMFScala.scala*

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, 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 access a model from 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. 5.2 together with their translations to Scala. We developed a small Scala library that makes interaction with Eclipse Modeling Framework slightly easier and enforces a few conventions.[5] In Table 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 Table 5.2 are slightly disturbing. Every time we access an Ecore collection we convert it to Scala. These inessential conversions are caused merely by impedance of 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 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 decid-

[5] See source: scala/src/main/scala/mdsebook/scala/EMFScala.scala. You can recreate a similar facility for any mainstream programming language in a few hours.

able. 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. 60), 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:

```
implicit class ImpliesExtension (a: Boolean) {
  def implies(b : => Boolean) = !a || b
}
```

Figure 5.8: *A Scala extension adding the implies operator to the* `Boolean` *class*

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 very well as an operator. Any language equipped with an extension mechanism will allow adding a new operator 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, without implication, one of their decision branches becomes constant:

```
inv[Model] { M =>
  M.getMachines.asScala.forall { m1 =>
    M.getMachines.asScala.forall { m2 =>
      if (m1!=m2)
        m1.getName!=m2.getName
      else
        true } } }
```

Figure 5.9: *An example of an overly verbose if-then-else expression with a constant branch*

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 may be able to eliminate an implication entirely. In Constraint **C3** the implication between predicates has been replaced by a set membership test. In general 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 found other ways of expressing the precondition, 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 commuting 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. 5.2 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 (Sect. 6.8) 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 Table 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 defined 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 languages. 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 page 64. The abstract syntax ADT code is available at fsm.scala/src/main/scala/dsldesign/fsm/scala/adt/Pure.scala

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 a just a task in a larger system endeavour, 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 around 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 Table 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 the Eclipse Modeling Framework, 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 are using 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.[6] 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 Table 5.3, Scala and Java (sic!) use conversions between legacy collections and other types. In Scala, as mentioned above, the standard library provides suitable higher-order API. In Java, the standard lists lack such 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.

[6] https://www.eclipse.org/modeling/emf/

Scala: Constraint C2 repeated from Table 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/mdsebook/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 bring DSLs to 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 less-than operator as implication, in a "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 is the most verbose of the used languages. Its collection API is rather underdeveloped. We cast lists to streams, in order to access quantifier functions. The constraint could be made more concise using a functional programming library.

fsm.java/src/main/java/mdsebook/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/mdsebook/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/mdsebook/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/mdsebook/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 Table 5.2 implemented in nine programming languages for comparison.

The Python implementation is based on the pyecore library.[7] The JavaScript implementation is based on the ecore.js library.[8] What is interesting 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),[9] 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, observing a standard contract between the validator and other components in the tool chain. This contract is important, as the validation logics 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 is typing. Obviously, the user creating a model should not see her model changed by the validation logics while typing.

In all nine languages, we have used anonymous functions to represent constraints, with the context element is 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 dropping the set/get prefixes on accessor 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 property 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`), (iii) and a quantifier to establish the result (`all/All/all`). The relational encoding exploits a classic result from data-base theory due to Codd (1972) that relational queries and first-order predicates over data

[7] https://github.com/pyecore/pyecore

[8] 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.

[9] https://github.com/NMFCode/NMF

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, both in 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. 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. Exactly, for 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 Table 5.2 in your favourite 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 the abstract syntax model supports establishing constraints? Are there any design issues with it? How well does your programming language support writing constraints? Consider 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?

5.4 Specialized Constraint Languages for Modeling

General purpose languages are hard to beat where it comes to ease of integration with the rest of your project, 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. If you are meta-modeling in Ecore or UML then 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 logics given 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. 5.2. OCL allows defining new functions, including recursive functions. It is, thus, more expressive than first-order logics. 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. 159).

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. 58, 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 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 simplicity of the formulation. The same advantages are observed in Line 14 for **C2**. Compare how these constraints have been written in general purpose programming languages in Table 5.3.

The OCL standard library provides a rich set of logical and set operations, the usual collection manipulation operations, and all Ecore meta-model types are directly accessible. We gather the operations used in our examples along with a few extras in Table 5.4. The `closure` operation deserves a longer discussion. We used `closure` in the implementation of **C5** in Line 18. It computes 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

```
 1 package fsm : _'mdsebook.fsm' = 'http://www.mdsebook.org/mdsebook.fsm' {
 2   abstract class NamedElement {
 3     attribute name: String[1];
 4   }
 5 
 6   class Model extends NamedElement {
 7     property machines: FiniteStateMachine[*|1] { ordered composes };
 8     invariant C1: machines->forAll (m1, m2 | m1 <> m2 implies m1.name <> m2.name);
 9   }
10 
11   class FiniteStateMachine extends NamedElement {
12     property states#machine : State[+|1] { ordered composes };
13     property initial : State[1];
14     invariant C2: states->forAll (s1,s2 | s1 <> s2 implies s1.name <> s2.name);
15     invariant C3: states->includes (initial);
16     invariant C5:
17       let reachable: Set(State) =
18         initial->closure (s | s.leavingTransitions->collect (t: Transition|t.target))
19       in states->forAll (s | reachable->includes(s));
20   }
21 
22   class Transition {
23     property target: State[1];
24     property source#leavingTransitions : State[1];
25     attribute input: String[1];
26     attribute output: String[?];
27     invariant C4: source.machine = target.machine;
28   }
29 
30   class State extends NamedElement {
31     property leavingTransitions#source : Transition[*|1] { ordered composes };
32     property machine#states : FiniteStateMachine[1];
33   }
34 }
```

Figure 5.10: *Constraints C1-C5 in OCL embedded in a textual representation of the meta-model of Fig. 3.1.*

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. 5.2.

Exercise 5.11. Implement Constraint **C6** from page 157 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/mdsebook/scala/EMFScala.scala as an example. Reimplement constraints C1 and C2 using the new quantifiers.

`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 of `n` element tuples 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 subtype (observes generalization).

Table 5.4: *An abridged reference 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*

```
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 property* `isInitial` *added to the* `State` *class using OCL.*

Often when writing constraints, you discover 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 could use a `let` expression to introduce a function computing the new derived property. 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 property should be accessible from many places? Also from other constraints?

OCL allows defining *derived properties*, to address this use case. Derived properties 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 a property of a state machine object (`initial`). There is 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 property (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 5.23 explores this connection.

> **Exercise 5.13.** Rewrite constraints **C3** in the context of `State` class using the `isInitial` property.

There are several independent implementations of OCL. To write this chapter we used the so called *Pivot OCL* implementation from the Eclipse MDT project.[10] 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.[11] 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 (Object Management Group, 2010) 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. 5.3. Alloy, and a few related languages,

[10] https://wiki.eclipse.org/OCL/Pivot_Model, seen 2020/05/11
[11] https://www.eclipse.org/epsilon/doc/evl/

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. Syntactically, Alloy (D. Jackson, 2006) is a textual structural modeling language that corresponds (roughly!) to the 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 signatures 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. 5.2. We have seven signatures in Fig. 5.12: `Name`, `Label`, `NamedElement`, `Model`, `FiniteStateMachine`, `State`, and `Transition`. The first two, `Name` and `Label`, are introducing 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, no contents except their own identity, and use them to label named elements and transitions. This will allow 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 its specializations only. A named element has a single property 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. To build an efficient mental model of Alloy, it is important to understand, that even though `name` is written syntactically as if this was a field property 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 property 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 (Gamma et al., 1995), 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.

```
1 sig Name { }
2 sig Label { }
3
4 abstract sig NamedElement { name: one Name }
5
6 one sig Model extends NamedElement {
7   machines: some FiniteStateMachine }
8
9 sig FiniteStateMachine extends NamedElement {
10   states : set State,
11   initial: one State }
12
13 sig State extends NamedElement {
14   leavingTransitions: set Transition,
15   machine           : one FiniteStateMachine }
16
17 sig Transition { target: one State,
18                  input : one Label,
19                  output: lone Label,
20                  source: one State }
21
22 fact { Model.machines = FiniteStateMachine }
23 fact { FiniteStateMachine.states = State }
24 fact { State.leavingTransitions = Transition }
25 fact { machine = ~states }
26 fact { source = ~leavingTransitions }
27
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.*

The signature has a property `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 "more than one" (`1..*`). Observe that `Model` extends `NamedElement`, so it also has the property `name` inherited from `NamedElement`. The other signature definitions follow analogously. The keyword `set` (lines 10 and 14) means "any number," 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 first twenty lines of Fig. 5.12. Notice, that a single state is shared by two

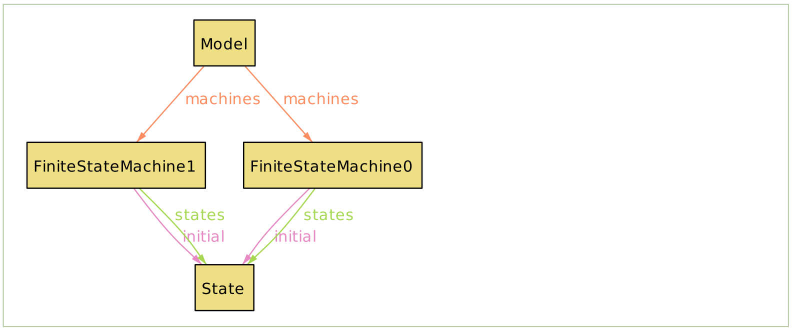


Figure 5.13: An instance of a FSM with two machines sharing a single state. This instance has been generated by Alloy analyzer for the first twenty lines of the model in Fig. 5.12

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. This together disallows objects that float free, outside a partonomy. To understand the syntax of these constraints, note that `Model.machines` computes (database-like) join of the set of models with the `machines` relation, ultimately resulting in a set of all machines that are related to a model. The constraint requires that this set is the same as the set of all finite state machines (l. 22). 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 (lines 25–26). The first constraint says that the `machine` relation is the inverse of the `states` relation. The second line 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 constraints 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 sets 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 relation created by joining the transition objects with machine objects via source states and via target states is 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 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 want to use Alloy's transitive closure construction. See lines 33–34 in Fig. 5.12. This is the most concise and the most direct presentation of this constraint 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 a 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 two 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

$$\texttt{State} \times \texttt{Transition} \times \texttt{State} \quad . \tag{5.24}$$

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.

We shall formalize this. Let the bow tie symbol (⋈) denote a "forgetting" join:

$$R \bowtie Q \quad \equiv \quad \{(r,q) \mid \text{there exists } p \text{ such that } (r,p) \in R \text{ and } (p,q) \in Q\} \quad . \tag{5.25}$$

Then the reflexive transitive closure of `leavingTransitions.target` can be described as:

$$\texttt{*(leavingTransitions.target)} = \bigcup_{i=0}^{\infty} \underbrace{\texttt{leavingTransitions.target} \bowtie \cdots \bowtie \texttt{leavingTransitions.target}}_{i \text{ times, forget internal columns}} . \tag{5.26}$$

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. 166. 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 a 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), `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 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 quantifier. To add a constraint to context of a signature place a Boolean predicate in its own braces directly after the closing the signature, without any keyword, as in: `sig ... { properties 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. D. Jackson (2006) 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 such case, just add an instantiation constraint for the type of the tested element. For example, if you 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 by need. This also makes the tools faster. The bound is specified as part of the query to the analyzer.

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 sometimes helps to understand 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, or any of the GPLs we 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 to Alloy syntax, and use Alloy tools to evaluate constraints on them. Once instances are

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*

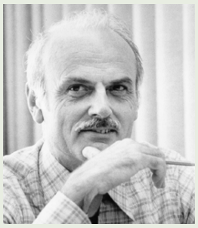
found, the Alloy output can be parsed and translated to whatever technical space you need. Since Alloy is a solver, not an evaluator, a whole range of more powerful checks can be implemented than using other languages.

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 for 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'`.

Constraints as a Modeling Paradigm


Edgar Frank Codd
(1923–2003)

Constraints are the basis of a very useful paradigm in modeling, exercised at its full in tools from the *Constraint Programming* community (Dechter, 2003; Rossi, Beek, and Walsh, 2006). Many constraint languages derive from first-order logics and relational algebras, exploiting the result of Edgar Codd that first-order logics adequately capture 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 generate-able 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 logics, 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.

Exercises Exercise 3.34 (p. 91) and 4.57 (p. 150) explore parsing of Alloy outputs.

This method has been used in research for extremely many DSLs. Over years, researchers have been exploiting 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.

5.5 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

Consider not-defining the static semantics at all! This may sound crazy, especially after reading four sections arguing exactly the opposite! However, side stepping constraints and type system definition (Sect. 6.8) is often a natural choice. 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

```
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."*

reporting, are often nice-to-have features that can be designed much later, when the scope and design of the language are stabilizing, and the rest of the tool chain is maturing.

For some minimalistic languages, developed on a tight resource 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* (Lämmel, 2018). Imagine, for instance, that we generate C code from finite state machine 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 the one 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 for your users, you may choose not to implement the constraint validation at all!

A general purpose or a specialized constraint language? Should I follow the advice of Sect. 5.3 or of Sect. 5.4? Should I use an external tool like Alloy? Shall I design all constraints in basic mathematics first, like in Sect. 5.2, and only then rewrite to a programming language?

The decision between specialized and general purpose languages hinges on the trade-off between the 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 use 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 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 logics as a software development mechanism, but as a way to introduce you to constraints thinking.)

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.

Maximize the diagram, minimize the constraints. A class diagram is a constraint system itself, just of limited expressiveness. As already discussed in Sect. 5.2, 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 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 use of 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 run-time 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 required 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 logics-based formalism.

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 Figs. 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 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 noticed 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 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

Keep constraints declarative, as close to natural language as possible. There is clearly a correspondence between requirements written in English and in formal logics and programming languages. To void obfuscating this correspondance, 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. 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.

```
1   val C2_GOOD = inv[FiniteStateMachine] { self =>
2     self.getStates.forAllDifferent { _.getName != _.getName } }
3
4   def C2_BAD (m: FiniteStateMachine): Boolean = {
5     var it1 = m.getStates.iterator
6
7     while (it1.hasNext) {
8       var s1: State = it1.next
9       var it2 = m.getStates.iterator
10
11      while (it2.hasNext) {
12        var s2: State = it2.next
13        if ( s1 != s2 && s1.getName == s2.getName)
14            return false
15      }
16    }
17    return true
18  }
```

Figure 5.16: ***Top:*** *(l. 1–2) Our favourite declarative formulation of* ***C2*** *in Scala.* ***Bottom:*** *(l. 4–18) A purposefully verbose and lengthy imperative formulation of* ***C2****, typical of inexperienced constraint writers. Avoid this! Both examples are written in Scala, but the problem is easily exemplified in most languages.*

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 for the user. Specialized constraint languages are designed to avoid such mistakes. Exercise extra care when using a GPL.

The only exemption from 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 instances, 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.

Avoid top-level conjunction. It might be tempting to combine several different aspects in a single constraint using logical conjunction at the top level. This should be avoided. One constraint should capture one English sentence, as simple as possible. There is rarely advantage to creating compound constraints. Small atomic 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.

```
inv[Student] { self =>
    !self.getName.isEmpty && self.getAge >= 18 }
```
Error: An empty name or the age 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.*

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 capabilities. The figure below 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.

```
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*

The use of `allInstances` should be avoided in favor of a good context class for the constraint. The 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.

Refactor constraints to the optimal context. Constraints written in a suboptimal context tend to contain long navigations and (too many) quantifiers. In the following example, the class `Model` is a much worse choice for the context of Constraint **C3** than the `FiniteStateMachine`, which we have chosen 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. 164

```
inv[FiniteStateMachine] { self =>
  self.getStates.contains (self.getInitial) }

inv[Model] { self =>
  self.getMachines.forall { fsm =>
    fsm.getStates.contains (fsm.getInitial) }
}
```

Exercise 5.17. Besides long navigations and excess quantifiers, inability to navigate to the set of instances needed in a constraint is also an indicator of a possible mistake in selection of the context type. If, you need to resort to tricks like using `allInstances` (in OCL) to work around such problems, there is good chance the context is incorrect. To appreciate this issue attempt to re-write the Constraint **C3** to the context of class `NamedElement` of the meta-model in Fig. 3.1.

It takes experience to select the right context. Check sequences of universal quantifiers if any prefix of them can possibly be avoided, by shifting the context to the type one of them is ranging 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 revisiting the meta-model design adding explicit or derived associations. A constraint relating very remote classes is an indicator that there is a missing association in the model.

Avoid verbose Boolean constructs, especially using true and false. Programmers inexperienced in functional style, that is so strongly exercised in constraints, tend to write expressions that are suboptimal in subtle but annoying 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.

Do not write	Write instead
`if (e) true else false`	`e`
`if (e) false else true`	`!e`
`if (e1) e2 else false`	`e1 && e2`
`if (e1) true else e2`	`e1 || e2`
`if (e1) e2 else true`	`e1 implies e2`
`if (e1) true else e2`	`!e1 implies e2`
`e && true`	`e`
`false || e`	`e`
`!e1 || e2`	`e1 implies e2`
`!(!e)`	`e`
`return e;`	`e`

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

5.6 Quality Assurance and Testing for Static Semantics

Ratiu, Voelter, and Pavletic (2018) report that they found many errors in their extensive collection of DSLs, despite being relatively sure that the front-end is solid (front-end errors are very visible to the users of the language, so developers spot them quickly and fix them early.) The importance of testing is growing while we proceed into the later stages of the language implementation, as the bugs get more intricate and harder to spot for the developers. They literally reported that the automated random machinery has found dozens of problems immediately.

2. Scenario based testing (consistency, stability, coverage)

Exercise 5.18. Create an instance of the meta-model in Fig. 3.1 that violates the constraint in Fig. 5.18 (for simplicity, focus on either the first or the second variant of the constraint). Test that you indeed violate the constraint, by validating it on the created instance, using your favourite constraint language infrastructure.

Show constraint C2 (all state within the same machine must have distinct names). We show the scala version again (and we may recall that the other table has shown this in XXX langagues.)
positive test-cases: test-00, test-07, test-09 negative test-case: test-08 (they are all in mdsebook.fsm/testfiles)
In the section we might show how these are called (one positive, one negative), extracted from ConstraintsSpec.scala

Figure 5.20: A positive and a negative test case for a constraint. Different than the exercise, one of our 6 constraints

Fig. 5.20 uses test cases in abstract syntax (stored as xmi files). We drawn them as instance specification diagrams, so that it is easier for you to read them. But link to the xmi file for the real thing. Show how to load. It is important that we can test both on concrete and abstract syntax. It is quite common in language implementation that the static semantics is implemented before (or in parallel) to the parser. By using abstract syntax to test static semantics we decouple the two modules in the implementation.

Figure: generators for state machines #code They are in mdse-book.fsm.scala/src/main/scala/mdsebook/fsm/scala/Generators.scala I think we should show both kinds, but not the convenience methods (one should be enough). So likely we are ending with two figures. Shorten drammaticaly. Also we can mention in the text that the first thing we do with this is fuzzing generators and fuzzing constraints. This already finds a lot of basic structural problems, especially good for generators that may violate ecore constraints, and for constraints that might navigate over null references.

5.7 Static Semantics in the Language Conformance Hierarchy

OCL is a DSL itself. Its concrete syntax is specified using a context-free grammar, and its abstract syntax is specified using a meta-model. The core part of this meta-model is shown in Fig. 5.22. Fig. 5.23b shows how this metamodel fits the general layered architecture of meta-modeling that we have shown previously.

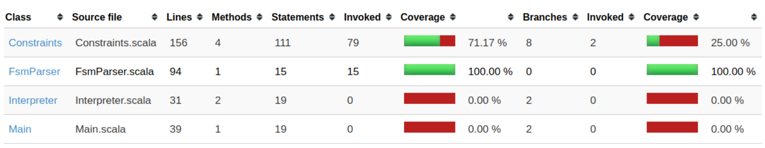
Fig. 5.23a explains another point: that OCL Language and OCL constraints semantically are similar to Ecore itself, so one can also draw them one level higher. This is because the constraints written at a given level constrain instances one level below. Thus, meta-model constraints have a similar semantic effect to the meta-models (M2), and therefore OCL is a specification language at M3, similarly to Ecore.

Further Reading

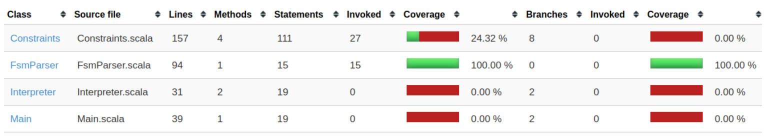
Our formalization of class diagrams using logics was arguably a bit hasty. For example, we have not formalized the link between instances of a formula and the logical constraints. There are many research papers on this topic. We mention a few that we know first hand, as authors. A small and very concise definition can be found in the work of Fahrenberg et al. (2014), or in the definition of formal semantics for Clafer (Juodisius et al., 2019). 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 on the same problem of classes vs instances, is often found in categorical approaches, for instance in the work of Bak et al. (2016).

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 the graph algorithms, see for example the book of Valiente (2002). It can be used to uncover the core dependence structure from a logical description, for example in model synthesis (She et al., 2014; Czarnecki and Wąsowski, 2007).

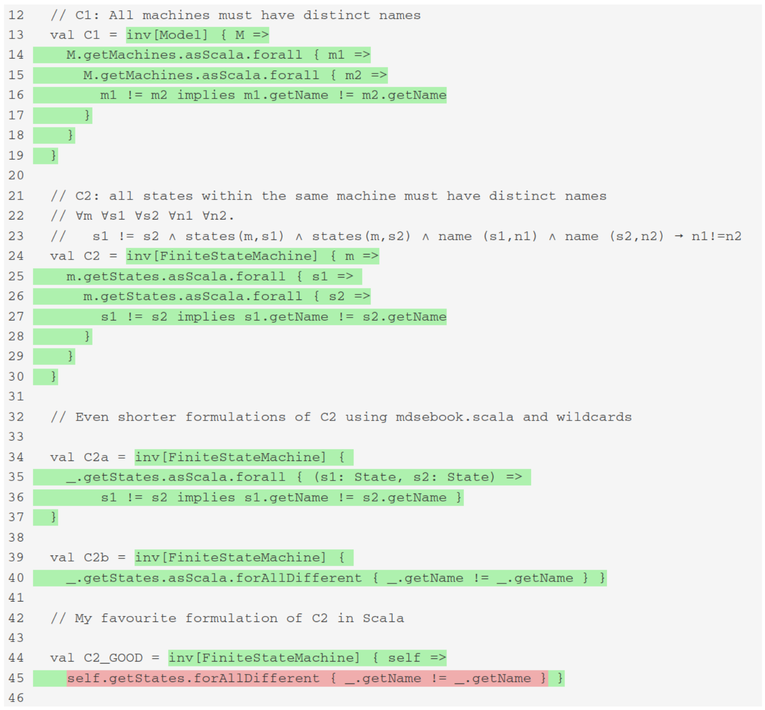
The standard reference on of OCL is the book of Warmer and Kleppe (2003). Chapter 3 presents guidelines for writing constraints (see esp. Section 3.10 on tips and tricks). A more concise, but still comprehensive, presentation can be found in a 30 page long tutorial paper of Cabot and Gogolla (2012). Another tutorial-like re-



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 %



Class	Source file	Lines	Methods	Statements	Invoked	Coverage		Branches	Invoked	Coverage	
Constraints	Constraints.scala	157	4	111	27	🟩🟥	24.32 %	8	0	🟥	0.00 %
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 } }

```

This is the coverage of scenario tests (but the coverage for fuzzing is the same). The remaining 30% we have simply not tested (all sorts of examples that we do not consider) and the 25% branch coverage is misleading, because there are very few branches in the file (and then they are mostly in the code we are actually not testing - branching coverage does not measure the branching induced by functional expressions).
It is probably not useful to compare fuzzing from scenario on these small constraints (although this is telling us that our scenarios are well sellected, for a larger project, it is hard to manually reach this level).

Figure 5.21: TODO: Figure: Show a coverage screenshot for our constraints in Scala. #code

Figure: consistency/validity laws. The code is in ConstraintsFuzzSpec.scala in the bottom, and everything works fine now.

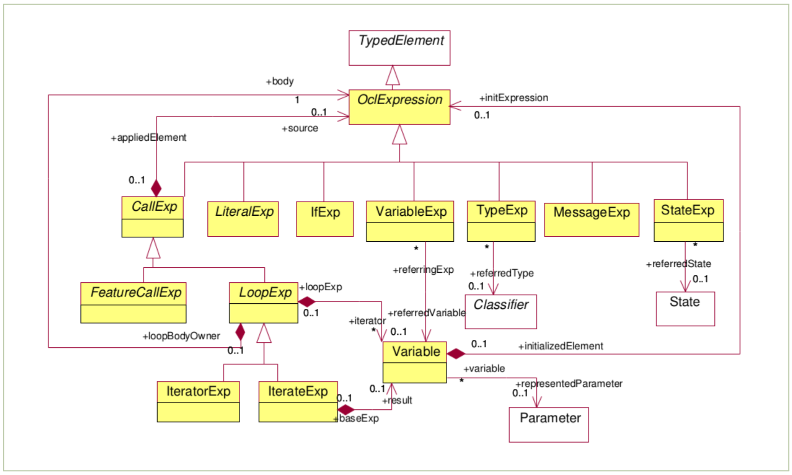
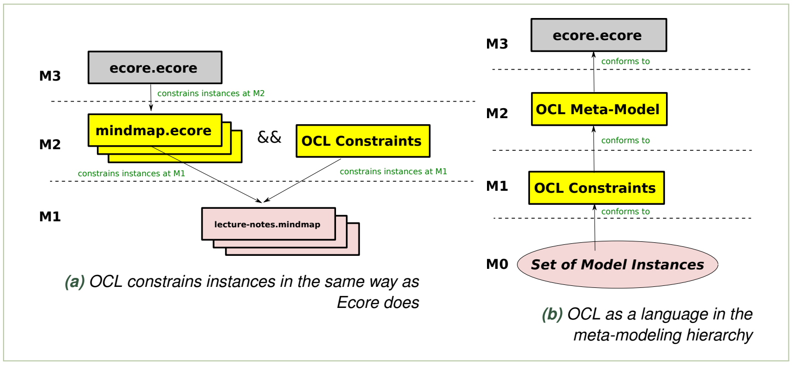


Figure 5.22: Core OCL Meta-Model from the OCL Specification (Object Management Group, 2010)



(a) OCL constrains instances in the same way as Ecore does

(b) OCL as a language in the meta-modeling hierarchy

Figure 5.23: Two views on OCL in the meta-modeling hierarchy

source are the slides of a course on OCL by Demuth (2009). The current official OCL specification can always be found at http://www.omg.org/spec/OCL/Current/. 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 (D. Jackson, 2019). His book (D. Jackson, 2006) and the journal paper on Alloy (D. Jackson, 2002) contain short and interesting critiques of OCL. He contrasts OCL with Alloy, emphasizing the difference between the relational and logical style. Besides Alloy, other modeling languages take constraints to the extreme, most notably Formula (E. K. Jackson and Schulte, 2013) and Clafer (Bak et al., 2016). 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 to specify part-of relationships directly like in class diagrams.

Additional Exercises

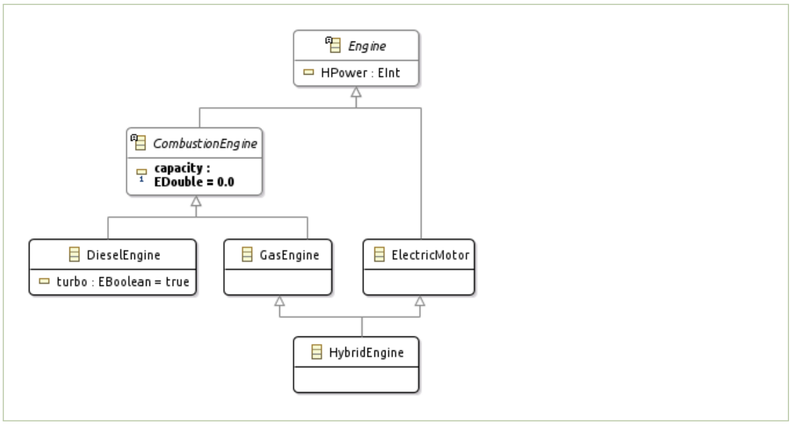


Figure 5.24: An example generalization hierarchy of car engine designs

Exercise 5.19. Which of the following first-order sentences hold for the generalization hierarchy shown in Fig. A.3?

a) $\forall x.\,\mathsf{HybridEngine}(x) \to \mathsf{CombustionEngine}(x) \wedge \mathsf{ElectricMotor}(x)$?

b) $\forall x.\,\mathsf{DieselEngine}(x) \to \mathsf{ElectricMotor}(x)$?

c) $\forall x.\,\mathsf{DieselEngine}(x) \to \mathsf{CombustionEngine}(x) \wedge \mathsf{ElectricMotor}(x)$?

d) $\forall x.\,\mathsf{CombustionEngine}(x) \wedge \mathsf{HybridEngine}(x) \to \mathsf{ElectricMotor}(x)$?

Exercise 5.20. Recall the OCL higher order function `isUnique` (see Table 5.4). Does the programming language you use for DSL implementations support this function? If yes, what is its name and type? Are there differences from OCL? If no, implement the function yourself, and use to solve Exercise 5.25c.

Exercise 5.21. Another common pattern of binary universal quantification is the so called *all-different* quantifier, where a property is enforced for *all pairs of different elements* in a collection. Both Constraint C1 and C2 are of this form. Implement `forAllDifferent` in a language of your choice and use it to simplify C1 and C2 further, by eliminating the precondition and implication. Our code repository shows Scala implementations in scala/src/main/scala/mdsebook/scala/EMFScala.scala as an example.

Exercise 5.22. Recall the `closure` operation from OCL that computes a 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 language of your choice. Refactor the implementation of Constraint **C5** in Table 5.2 to use the new operation.

Exercise 5.23. Derived properties can be implemented using extension methods in general purpose programming languages, including Scala, Kotlin, Groovy, Xtend, C#, and F#. Implement `isInitial` (Fig. 5.11) as an extension method

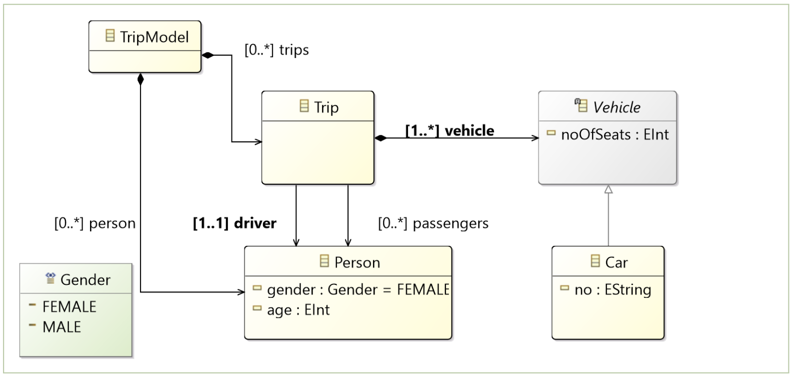


Figure 5.25: *A trip meta-model: describing simple travel arrangements for friends, see trip/model/trip.ecore*

in a programming language of your choice. Consider using value caching (for instance `lazy val` in Scala), to compute the derived value only the first time it is accessed. Retrieve it from a cache all the subsequent times.

Exercise 5.24. Recall the class types in the diagram of Fig. 5.1 in p. 153. Write the following commutativity constraints for this diagram (in logics, or in a programming language, or in a 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 5.25. Consider a simple model describing organization of trips Fig. 5.25. Write the following constraints for this meta-model:

a) The vehicle associated with the trip needs to be large enough to accommodate all the involved passengers: for each trip, the number of passengers must be smaller or equal than the number of seats in the involved vehicle.

b) The driver is included on the passenger list.

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 `TripModel` class, avoiding use of `allInstances` (if using OCL).

Exercise 5.26. Consider the naive meta-model in Fig. 5.26 describing a car.

a) For this meta-model, write the following constraint: *The driver seat in a car must be a seat in the same car* in the formalism of your choice.

b) Refactor the meta-model to make `driverSeat` a property in the `Seat` class (an attribute). How can you enforce the above constraint now?

c) Refactor the model of Fig. 5.26 to split the passenger seats (`3..6`) and the driver seat to two separate containments, `passengerSeats` and `driverSeat`. How can you now access the set of all seats? Implement a derived property `seats` that is a union of values of the two new attributes. Use derived properties if writing in OCL; extension methods, or protected code blocks in other languages.

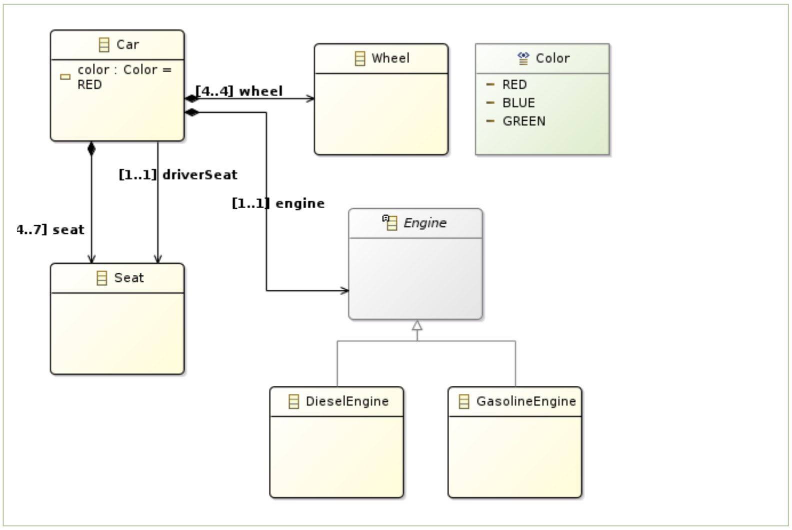


Figure 5.26: A simple model of structural components of a passenger car.

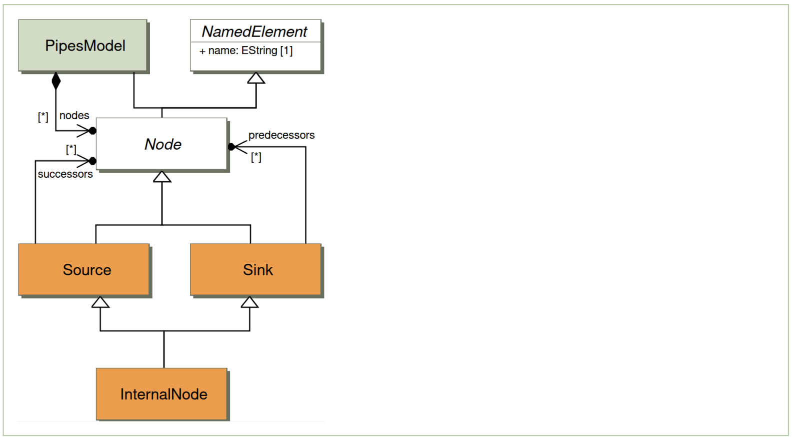


Figure 5.27: A meta-model for the core part of a flow-based web composition language.

Exercise 5.27. Figure 5.27 presents a simple meta-model of a flow language for creating web mash-ups.[a] 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). Figure 5.29 shows a constraint for this meta-model. Does the instance shown in Fig. 5.28 satisfy this constraint? Does it satisfy the meta-model constraints? If yes, argue how each constraint is satisfied. If not, point to the violating model element(s).

Exercise 5.28. For the model of Fig. 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.

[a] 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. The meta-model is available from the book's source project pipes with some constraints in pipes.scala.

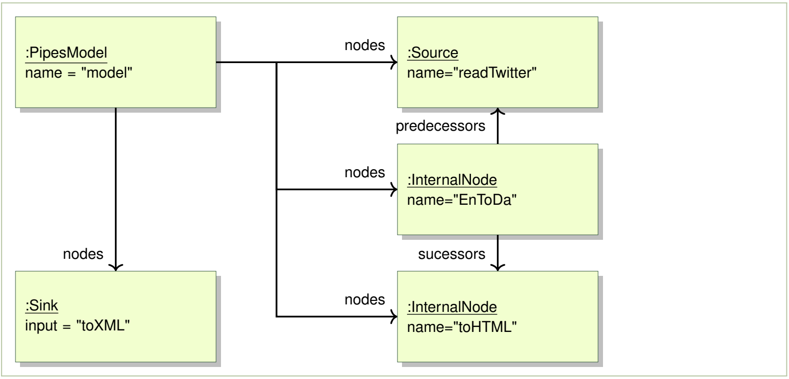


Figure 5.28: *An example instance for the meta-model of Fig. 5.27*

```
// 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
```

Figure 5.29: *A class cardinality constraint for Pipes in Xtend and Scala*

Exercise 5.29. 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. 5.30.[a] The models are available in Ecore XMI format from the book repository in the project named printers. Some constraints are shown in the project printers.scala. Write the following constraints for the meta-model in Fig. 5.30:

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 satisfies the constraint and verify by running the validation to see if 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. These two constraints are not identical. Can you explain the differences?

Exercise 5.30. Consider a new meta-model `T2.ecore` in Fig. 5.31. Write the following constraint in the context of the printer pool: *Each Printer pool with a fax, must have a printer, and each printer pool with a copier must have a scanner and a printer.*

[a]The printing example and the exercises are inspired by the submission for the standardization process of *Common Variability Language* within the Object Management Group (Submitters and supporters, 2012). This part of the standard proposal have been prepared by Krzysztof Czarnecki and Kacper Bąk.

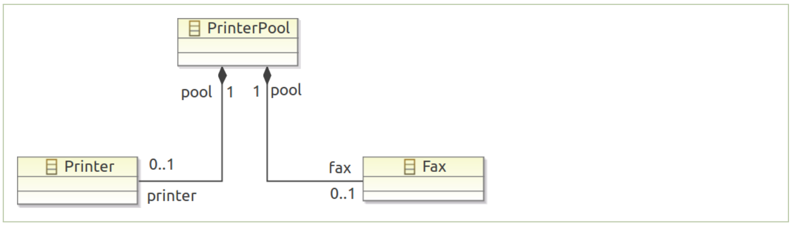


Figure 5.30: An example printer pool model `T1`

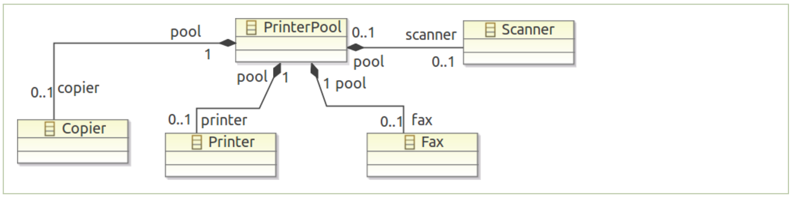


Figure 5.31: Class diagram `T2` showing a printer pool with scanners and copiers

Create an instance of the above model that satisfies the constraint and test if this is indeed the case. Create an instance of the above model that violates this constraint and verify that this is indeed the case, by running the constraint and checking that it fails.

Exercise 5.31. Consider the class diagram `T3` in Fig. 5.32. Write the following constraint in the context of the class `PrinterPool`: *PrinterPool's minimum speed must be 300 lower than its regular speed*. Validate the constraint with a positive and a negative instance.

Exercise 5.32. Consider the class diagram `T4` in Fig. 5.33. 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.

Exercise 5.33. Consider the class diagram `T5` in Fig. 5.34. Write the following constraint in the context of the class `Printer`: *A color capable printer pool contains at least one color capable printer.* Validate the result using a positive and a negative instance.

Exercise 5.34. Consider the class diagram `T6` presented in Fig. 5.35. Write the following constraints in the context of the class `PrinterPool`. Test both of these constraints using a negative and a positive instance for each.

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.*

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 nasty (second constraint in the context of Printer). Test all four cases with the same negative and positive instances as before. Discuss the differences that context changes introduce to constraints.

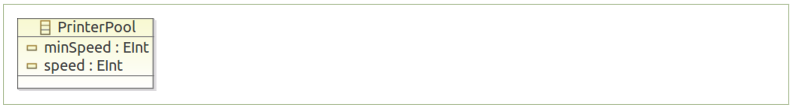


Figure 5.32: A class diagram `T3` *with attributes*

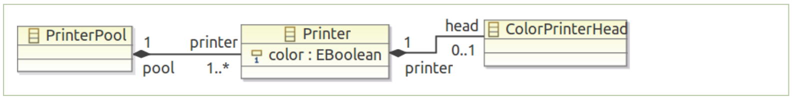


Figure 5.33: A class diagram `T4` *with* `ColorPrinterHead`

Exercise 5.35. For the diagram of Fig. 5.35 assert that *there is at most one color printer in any pool*. Test the constraint on a positive and a negative instance.

Exercise 5.36. Consider the over-simplified meta-model for SQL queries presented in Fig. 5.36 (projects sql and sql.scala).

a) Write a constraint that every `SelectQuery` selects from exactly one table, and all columns come from the same table. Write this constraint in the context of `SelectQuery`.

b) Write the same constraint in the context of the `Model` class.

c) Forget the above points–now a query can draw from several tables, but the tables used in a query must not have columns with the same names. Context: `SelectQuery`.

d) Forget the previous constraint. Write a constraint that all column names accessed in a single query are unambigous. So in a query using several tables, if a column is used then exactly one of the used tables has a column with the same name. Context: `SelectQuery`.

Exercise 5.37. In the meta-model of Fig. 3.20 (p. 87) 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 5.37 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 5.38. Write the following constraints over the instances of the Pascal's triangle meta-model of Fig. 3.18.

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 5.39. Figure 5.38 shows a simplistic meta-model for relational schema. The instances of this meta-model store primary keys in the `primaryKeys` collection, and the foreign keys in the `refersTo` collection.

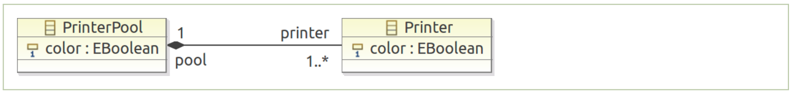


Figure 5.34: Class diagram T5 *with color printer pools and color printers*

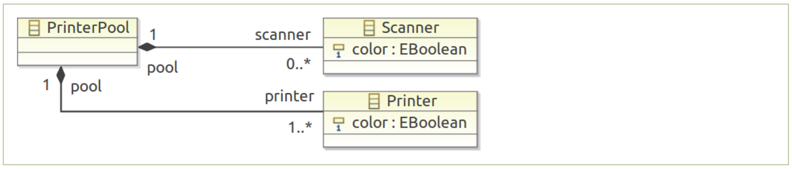


Figure 5.35: Colored scanners and printers in class diagram T6

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 from a table, and itself refers to a table. Test your constraint on a positive and a negative instance.

Exercise 5.40. Constraint the meta-model of Fig. 5.27 so that that from each `Source` instance one can reach a `Sink` instance via a series of `succesors` 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). Also, note that we need to arrive at a properly a `Sink` node, not in an `InternalNode` instance.

Exercise 5.41. For each of the following constraints indicate the *preferred context class* in the Pipes meta-model of Fig. 5.27.

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 5.42. We would like to constrain the instances of the pipes meta-model (Fig. 5.27) 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]
}
```

Unfortunately, testing shows that this constraint is incorrect. Analyze it and propose an improvement. Write the constraint in your chosen constraint language and validate it with a positive and a negative instance.

Exercise 5.43. We would like to constrain the instances of the pipes meta-model (Fig. 5.27), so that each sink has at least one predecessor. We do this using the following constraint written in the context of class `Sink`:

```
// (Xtend) Each sink has a non empty set of predecessors
def boolean constraint (Sink it)
{ ! predecessors.empty }
```

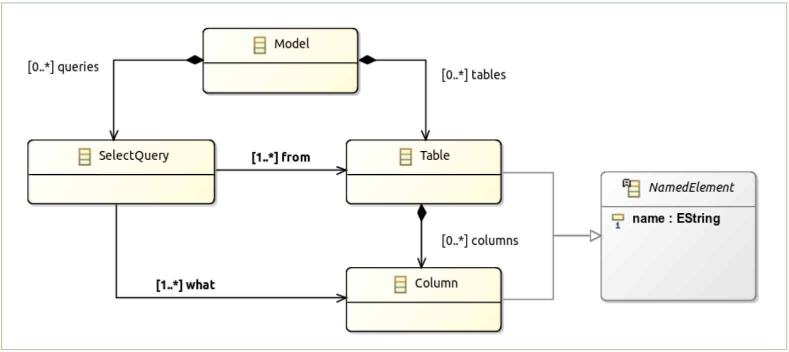


Figure 5.36: A meta-model for a very simple core of select queries in SQL

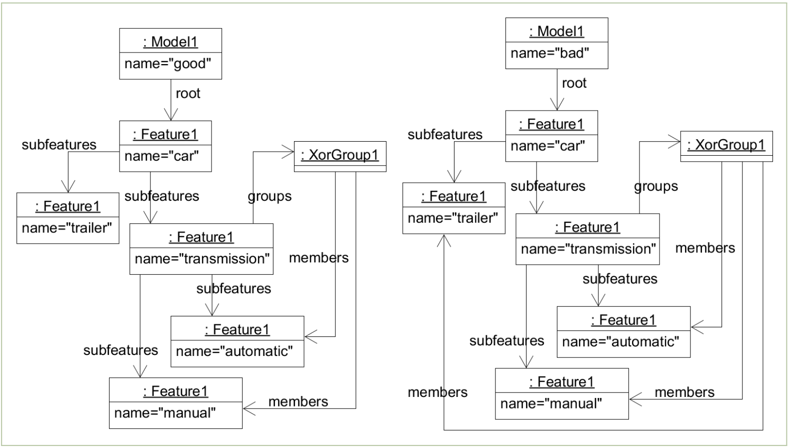


Figure 5.37: Examples of a good (left) and a bad (right) instance for Exercise 5.37 question A. The right one should be prevented by constraints.

Rewrite this constraint in the context of the `PipesModel` class in a constraint language of your choice.

Exercise 5.44. The pipes meta-model in Fig. 5.27 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 5.45. We want to constrain the Pipes meta-model of Fig. 5.27 with the two following constraints (enforced for all sinks and all sources, written in Xtend):

```
def public dispatch boolean constraint(Sink it) {

      ! predecessors.empty
}

def public dispatch boolean constraint(Source it) {

      ! successors.empty
}
```

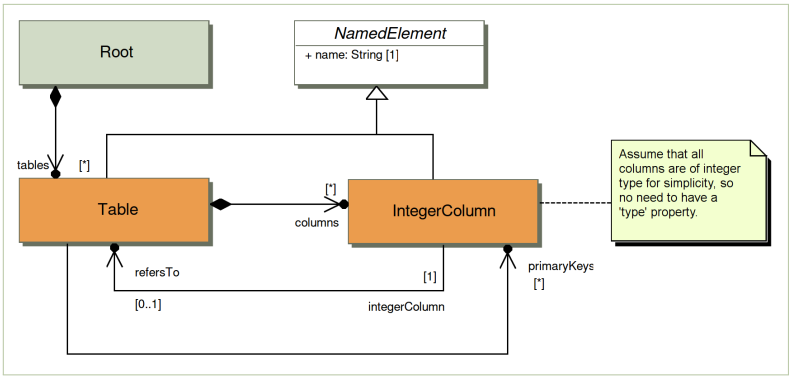


Figure 5.38: A simple meta-model for relational schema with tables and integer columns, primary keys, and foreign keys

Apparently these constraints are not needed, as they could be incorporated into the meta-model. Revise the meta-model to contain these constraints in the class diagram.

Exercise 5.46. We want to constraint the Pipes meta-model of Fig. 5.27 so that each instance model has exactly one source and exactly one sink:

```
// each model has exactly one source and exactly one sink
def public dispatch 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 to the constraint language used in your modeling environment.

Exercise 5.47. Recall the constraint presented in Fig. 5.29 for the model of Fig. 5.27. What are the suitable test-cases for testing the constraint? Discuss how would you select test-cases for this constraint (including example test-cases).

Exercise 5.48. Formulate the constraints from Exercise 5.34 in Java (say 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 5.49. Recall the key fragment of the Ecore meta-model presented in Fig. 3.25. 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 5.50. The micro Ecore meta-model of Fig. 3.25 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 for 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 5.51. Recreate the model of Fig. 5.26 in Alloy. Use Alloy analyzer to count how many possible configurations are possible. Then add the constraint from the Exercise 5.26a and repeat the counting. Reflect on this result. Has the number changed? Why?

References

Bak, Kacper et al. (2016). "Clafer: unifying class and feature modeling". In: *Software and System Modeling* 15.3, pp. 811–845. DOI: 10.1007/s10270-014-0441-1. URL: http://dx.doi.org/10.1007/s10270-014-0441-1.

Cabot, Jordi and Martin Gogolla (2012). "Object Constraint Language (OCL): A Definitive Guide". In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Vol. 7320. Lecture Notes in Computer Science. Springer, pp. 58–90. DOI: 10.1007/978-3-642-30982-3_3. URL: https://doi.org/10.1007/978-3-642-30982-3_3.

Codd, E. F. (1972). "Relational Completeness of Data Base Sublanguages". In: *Research Report / RJ / IBM / San Jose, California* RJ987.

Czarnecki, Krzysztof and Andrzej Wąsowski (2007). "Feature Diagrams and Logics: There and Back Again". In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, pp. 23–34. DOI: 10.1109/SPLINE.2007.24. URL: https://doi.org/10.1109/SPLINE.2007.24.

Dechter, Rina (2003). *Constraint Processing*. Morgan-Kauffman.

Demuth, Birgit (2009). *OCL (Object Constraint Language) by Example*. http://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf.

Fahrenberg, Uli et al. (2014). "Sound Merging and Differencing for Class Diagrams". In: *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Stefania Gnesi and Arend Rensink. Vol. 8411. Lecture Notes in Computer Science. Springer, pp. 63–78. DOI: 10.1007/978-3-642-54804-8_5. URL: https://doi.org/10.1007/978-3-642-54804-8%5C_5.

Gamma, E. et al. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

Jackson, Daniel (2002). “Alloy: a lightweight object modelling notation”. In: *ACM Trans. Softw. Eng. Methodol.* 11.2. http://doi.acm.org/10.1145/505145.505149, pp. 256–290.

– (2006). *Software Abstractions*. MIT Press.

– (2019). “Alloy: a language and tool for exploring software designs”. In: *Commun. ACM* 62.9, pp. 66–76. DOI: 10.1145/3338843. URL: https://doi.org/10.1145/3338843.

Jackson, Ethan K. and Wolfram Schulte (2013). “FORMULA 2.0: A Language for Formal Specifications”. In: *Unifying Theories of Programming and Formal Engineering Methods - International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, China, August 26-30, 2013, Advanced Lectures*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Vol. 8050. Lecture Notes in Computer Science. Springer, pp. 156–206. DOI: 10.1007/978-3-642-39721-9_4. URL: https://doi.org/10.1007/978-3-642-39721-9%5C_4.

Juodisius, Paulius et al. (2019). “Clafer: Lightweight Modeling of Structure, Behaviour, and Variability”. In: *Art Sci. Eng. Program.* 3.1, p. 2. DOI: 10.22152/programming-journal.org/2019/3/2. URL: https://doi.org/10.22152/programming-journal.org/2019/3/2.

Lämmel, Ralf (2018). *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer.

Lem, Stanisław (1967). *The Cyberiad: Fables for the Cybernetic Age*. Trans. by Michael Kandel. ISBN: 0156027593.

Object Management Group (2010). *OCL Specification version 2.2*. http://www.omg.org/spec/OCL/2.2/.

Ratiu, Daniel, Markus Voelter, and Domenik Pavletic (2018). “Automated testing of DSL implementations—experiences from building mbeddr”. In: *Software Quality Journal* 26.4, pp. 1483–1518.

Rossi, Francesca, Peter van Beek, and Toby Walsh, eds. (2006). *Handbook of Constraint Programming*. Elsevier.

She, Steven et al. (2014). “Efficient synthesis of feature models”. In: *Information and Software Technology* 56.9. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2014.01.012. URL: http://www.sciencedirect.com/science/article/pii/S0950584914000238.

Submitters and supporters (2012). *Common Variability Language. OMG Revised Submission*. URL: http://www.variabilitymodeling.org.

Valiente, Gabriel (2002). *Algorithms on Trees and Graphs*. Springer.

Warmer, Jos and Anneke Kleppe (2003). *The Object Constraint Language*. Addison-Wesley.

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, that is 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) 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. 5.3, in type systems we tend to use constraints that are directly executable (unlike the constraints in Alloy, which need to be solved). Consequently, the executable structural constraints, presented in the previous chapter, are the simplest possible type system—the 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 PyMC3.[1] PyMC3's interface can be seen as an *internal domain-specific language* (**??**) 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 $\sigma = y$.

$$\begin{aligned} &x = 0 \\ &y = 0 \\ &\mathcal{N}(\mu = x + z, \sigma = y) \end{aligned}$$

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 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 govern 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:

$$\begin{aligned}&x = \mathscr{U}(-1,1)\\&y = 1\\&\mathscr{N}(\mu = x+y, \sigma = y)\end{aligned}$$

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 a 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 (2002):

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.*

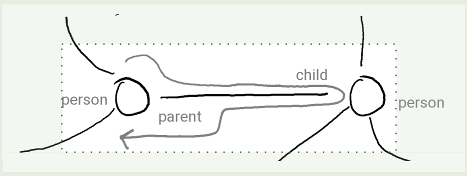
This abstract definition calls for a few explanations. First, a type system shall be *tractable*. The algorithm for establishing type correctness should be efficient, typically polynomial in the size of the input model. This is why we want to use only executable constraints. Theoretically, we could encode type correctness as constraints with free variables in a sufficiently rich logics, but solving them would be undecidable; far from polynomial-time. Type checkers typically use algorithms that infer types *inductively* by traversing the syntax tree (see the infobox on page 211).

A type checker is not a universal verification tool. It is constructed to prove limited concrete properties, to detect *selected errors*. These could be

[1] https://docs.pymc.io/

[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.

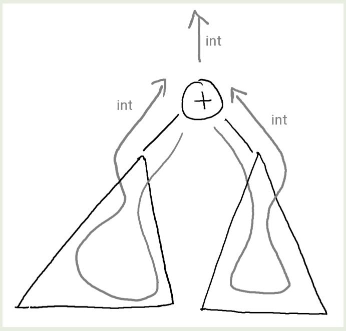
In contrast, the type of an arithmetic expression is not directly computable by just examining the instance object in question (an expression node) and a small number of its neighbours. Consider the rule for typing a binary addition expression: *The result of binary addition is an integer if both of 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.

errors in computation (like memory safety), but could also be errors in how 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 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 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 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 Scala implementation of an abstract syntax for `prpro` while Fig. 6.2 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. A declaration is an abstract type, with two concrete realizations. 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 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. Complex values have complex types, complex types are created from simpler types. Thus types are themselves

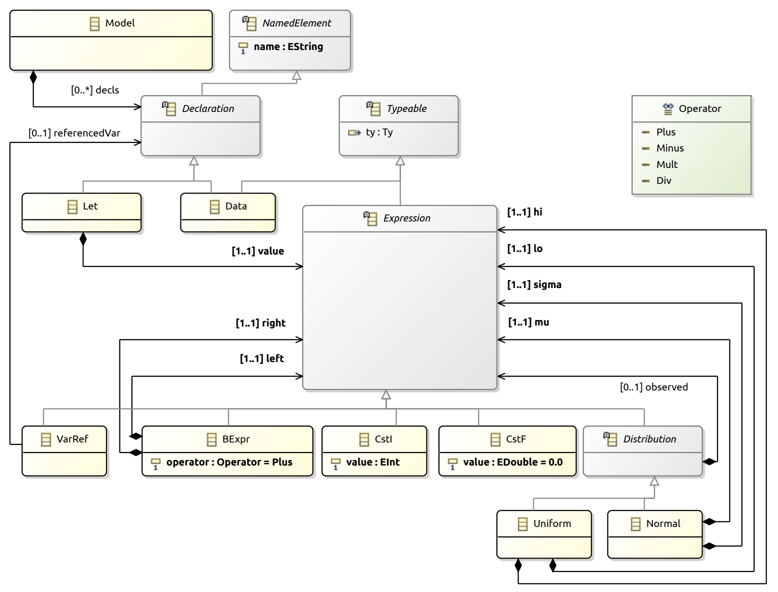
```scala
abstract trait NamedElement { val name: String }
trait Typeable {
    def getTy: Ty
    private[adt] def setTy (ty: Ty): Ty }

type Model = List[Declaration]
sealed abstract trait Declaration extends NamedElement
case class Let (name: String, value: Expression) extends Declaration
case class Data (name: String, ty: Ty) extends Declaration

sealed abstract trait Expression extends Typeable
sealed abstract trait Distribution extends Expression

case class Uniform (
  lo: Expression,
  hi: Expression,
  observed: Option[Expression] = None) extends Distribution

case class Normal (
  mu: Expression,
  sigma: Expression,
  observed: Option[Expression] = None) extends Distribution

case class BExpr (
  left: Expression,
  operator: Operator,
  right: Expression) extends Expression

case class VarRef (name: String) extends NamedElement with Expression
case class CstI (value: Int) extends Expression
case class CstF (value: Double) extends Expression

sealed abstract trait Operator
case object Plus extends Operator
case object Minus extends Operator
case object Mult extends Operator
case object Div extends Operator
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/Pure.scala

Figure 6.1: *Algebraic data types representing the abstract syntax of* `prpro`*, a simple probabilistic modeling language. See also Fig. 6.2*

expressions! Indeed, they are instances of another language—the *type language*. A type language is an abstraction of the *typed language*, here `prpro`. It follows the same core structure, but leaves out many details inessential for the property tracked by the type system. The process of typing is the processes 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 specify their syntax like for any other language: using a meta-model or algebraic data types. Figures 6.3 and 6.4 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



source: prpro.scala/src/main/scala/dsldesign/prpro/model/prpro.ecore

Figure 6.2: *An Ecore meta-model for* `prpro` *that follows a similar design to the ADTs in Fig. 6.1*

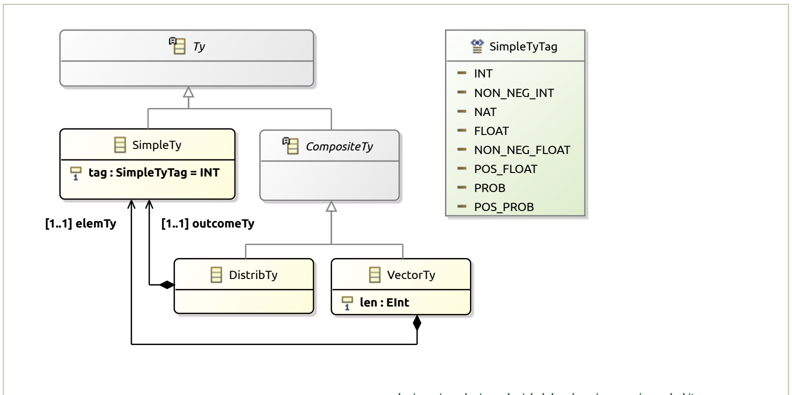
```
 1 sealed abstract trait Ty
 2
 3 sealed abstract trait SimpleTy extends Ty
 4 case object IntTy extends SimpleTy
 5 case object NonNegIntTy extends SimpleTy
 6 case object NatTy extends SimpleTy
 7 case object FloatTy extends SimpleTy
 8 case object NonNegFloatTy extends SimpleTy
 9 case object PosFloatTy extends SimpleTy
10 case object ProbTy extends SimpleTy
11 case object PosProbTy extends SimpleTy
12
13 sealed abstract trait CompositeTy extends Ty
14 case class VectorTy (len: Int, elemTy: SimpleTy) extends CompositeTy
15 case class DistribTy (outcomeTy: SimpleTy) extends CompositeTy
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/Types.scala

Figure 6.3: *An ADT representing a type language for* `prpro`*, a simple probabilistic modeling language. See also an Ecore definition of the type language in Fig. 6.4*

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. 6.3, these are defined in lines 3–11, in Fig. 6.4 in the enumeration SimpleTyTag.

Composite types are defined as case classes in Scala (lines 13–15 in Fig. 6.3) and as concrete classes in Ecore (the bottom part of the diagram in Fig. 6.4). `Prpro` includes binary expressions, distribution expressions, and



source: prpro.scala/src/main/scala/dsldesign/prpro/model/types.ecore

Figure 6.4: An Ecore meta-model for the language of types for `prpro` *that follows a similar design to the ADTs in Fig. 6.3*

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 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 vector's length is greater than one.

A `prpro` interpreter needs to distinguish distributions from simple values, as a different execution machinery is needed for them. Expressions involving distributions may exhibit several typing errors. For instance, we require that all vectors in `prpro` have constant fixed size, so we cannot use a distribution to specify a vector length. Similarly, we cannot add a distribution and a vector. We add another composite type: `DistributionTy`, which encapsulates the type of the elements a distribution generates. To keep the language simple we only allow distributions over simple 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. 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 of that step in this chapter.

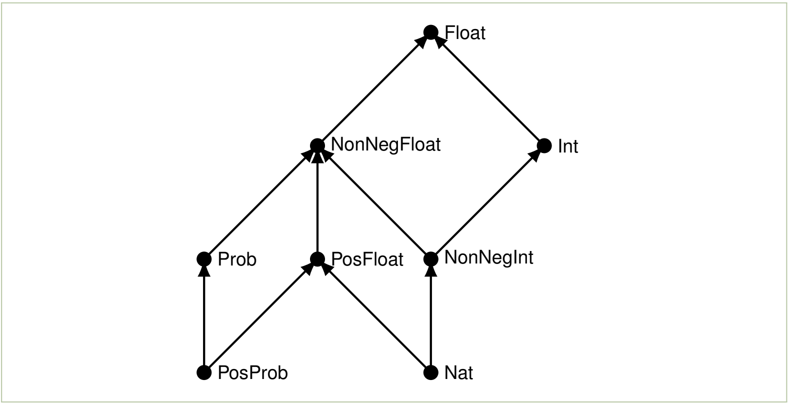


Figure 6.5: Sub-typing hierarchy for simple types in `prpro`*. There is an edge from type t_1 to type t_2 in the graph iff t_1 is a direct sub-type of t_2, written $t_1 \sqsubseteq t_2$*

6.3 Type Hierarchy

Most type systems define a notion of *refinement* or *substitutability*. Substitutability means that when a program (or 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 (Liskov and Wing, 1994). For example, a probabilistic model written in `prpro` shall allow to assign an integer number 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. 6.5. 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 subtype, 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. 6.5 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:

$$(\textsc{SSimple})\frac{t_1, t_2 \text{ simple} \quad t_1 \sqsubseteq^* t_2}{t_1 \leqslant t_2} \qquad (\textsc{SDist-1})\frac{t_1 \sqsubseteq^* t_2}{\mathsf{Distrib}(t_1) \leqslant \mathsf{Distrib}(t_2)}$$

$$(\textsc{SVect})\frac{l_1 \geq l_2 \qquad t_1 \sqsubseteq^* t_2}{\mathsf{Vector}(l_1, t_1) \leqslant \mathsf{Vector}(l_2, t_2)} \qquad (\textsc{SDist-2})\frac{t_1 \sqsubseteq^* t_2}{\mathsf{Distrib}(t_1) \leqslant t_2}$$

The rule SVECT 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 lengths 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.[3] The SDIST-2 rule handles the most controversial case in the type system of `prpro`: it admits refinement of a simple type by a distribtion type. We want to allow a probability distribution in a location where a simple value is normally used in an expression. A probability

[3]Here we are ignoring the fact that some values may have probability zero, so even if admitted by the type, might not actually be effectively realizable by the distribution.

```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)
    l1 >= l2 && (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/adt/Types.scala

Figure 6.6: *A Scala implementation of the inductive definition of the sub-typing relation from p. 217*

```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)
      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 6.7: *A fragment of the Java implementation of a sub-typing relation, rules* SSimple *and* SVect *from p. 217*

distribution instead of a simple value expresses uncertainty about this value. What does it mean for the type system? We allow to refine any simple type t_1 by a distribution generating values of any of its sub-type (SDIST-2).[4]

Exercise 6.1. Is `Float` the top type in the sub-typing hierarchy? In other words, is any other type in `prpro` a subtype 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 6.6 shows the implementation of the above rules in Scala, following the same order as in the formalization above. In Line 3,

[4]One can argue that distributions are a super-type of simple values, and 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. See also Exercise 6.16

we query a simple type for the set of its super types to test $\sqsubseteq$. Since the set of simple types is small and finite in `prpro`, we hard-coded their sub-typing relation as a property `superTys` of each simple type class. 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. 6.7. The structure and logics 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 to split computations 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 none of other rules 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 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. Can this types be added? 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 and a floating type 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 a 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. 6.5, 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 simultanously from t_1 and t_2, and continue until the two paths meet. Figure 6.8 shows that the lub of `PosProb` and `Nat` is the type of non-negative floats (`NonNegFloat`):

$$\texttt{PosProb} \sqcup \texttt{Nat} = \texttt{NonNegFloat} \tag{6.2}$$

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, this definition is going to be recursive. We are going to consider all possible

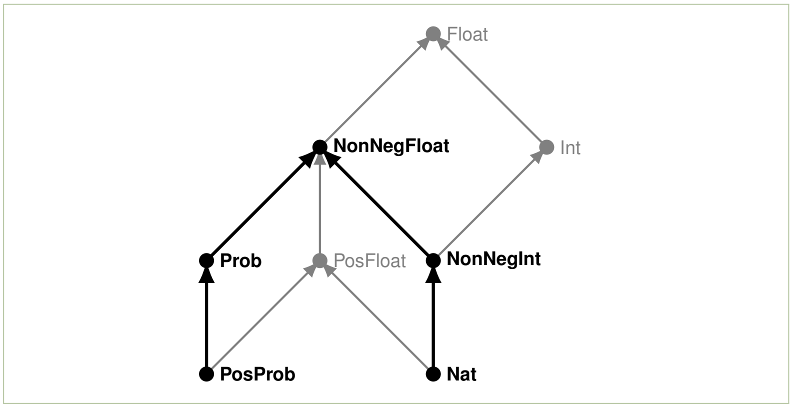


Figure 6.8: The least upper bound of types `Nat` *and* `PosProb` *is* `NonNegFloat` *(the proof construction shown in black)*

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. 6.5.

$$t_1 \sqcup t_2 = \begin{cases} t_1 \sqcup t_2 \text{ in Fig. 6.5} & \text{if } t_1, t_2 \text{ are simple} \\ \mathsf{Vector}(\min(l', l''), t' \sqcup t') & \text{if } t_1 = \mathsf{Vector}(l', t'),\ t_2 = \mathsf{Vector}(l'', t'') \\ \mathsf{Distrib}(t_1' \sqcup t_2') & \text{if } t_1 = \mathsf{Distrib}(t_1'),\ t_2 = \mathsf{Distrib}(t_2') \\ t' \sqcup t'' & \text{if } t_1 = \mathsf{Distrib}(t'),\ t_2 = t'' \text{ is simple} \\ t' \sqcup t'' & \text{if } t_1 = t' \text{ is simple},\ t_2 = \mathsf{Distrib}(t'') \end{cases} \tag{6.3}$$

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 (lub) of the elements of joined vector types. Compare these type transformations with the premises of rule SVECT. Clearly, the shortest vector that is 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 typable with the inferred element type of the vector.

Similarly, when joining two distribution types (the third case), we to join the element types and obtain a new distribution type that is the closest super-type as per rule SDIST-1. The final two cases deal with sub-typing along SDIST-2 and SSIMPLE (one side of the join is a distribution type, and the other side is a simple type). The cases join a distribution type with a simple type, resulting in the closest simple type above in the sub-typing hierarchy. Compare this with rule SDIST-2, which says that a simple type can be a super-type of 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 precomputed for any pair of simple types. However, pre-computing might be annoying in early design stages, when types

```
 1 private[adt] val topologicallySortedSimpleTys =
 2   List (NatTy, PosProbTy, NonNegIntTy, PosFloatTy, ProbTy, IntTy,
 3     NonNegFloatTy, FloatTy)

 5 def lub (t1: SimpleTy, t2: SimpleTy): SimpleTy =
 6   topologicallySortedSimpleTys
 7     .find { t => (t isSuperTypeOf t1) && (t isSuperTypeOf t2) }
 8     .getOrElse (null) // find always succeeds (an offensive null)

10 type Result[+T] = Either[ErrMessage,T]

12 def lub (t1: Ty, t2: Ty): Result[Ty] =
13     (t1, t2) match {
14       case (ty1: SimpleTy, ty2: SimpleTy) =>
15         Right (lub (ty1, ty2))
16       case (VectorTy (len1, ty1), VectorTy (len2, ty2)) =>
17         Right (VectorTy (len1 min len2, lub (ty1, ty2)))
18       case (DistribTy (ty1), DistribTy (ty2)) =>
19         Right (DistribTy (lub (ty1, ty2)))
20       case (DistribTy (ty1), ty2: SimpleTy) =>
21         Right (lub (ty1, ty2))
22       case (ty1: SimpleTy, DistribTy (ty2)) =>
23         Right (lub (ty1, ty2))
24       case _ =>
25         Left (s"An attempt to unify incompatible types: $t1, $t2")
26   }
```
source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/Types.scala

Figure 6.9: *A Scala implementation of join (the least upper-bound) for simple types (lines 5–8) and for composite types (lines 10–24) of* `prpro`

are changing a lot. Every time, you add or modify a type the precomputed map needs to be updated. To avoid this problem, we sacrificied efficiency of code 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 6.9 presents the implementation of type joining for both simple and composite types for the abstract syntax as a Scala ADT. In lines 1–3, 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. 6.5. (Why is `NatTy` first?) Lines 5–8 show the simplistic 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 8 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 12 onwards. The function attempts to compute a join of two types. It can fail, so the return value is wrapped into a `Result` value,

```
 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) throws TypeError
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     public Ty defaultCase (EObject t)
21     {
22       String msg = String.format (
23           "An attempt to unify incompatble types: %s, %s", t1, t2);
24       throw new TypeError (msg);
25     }
26   }

28   return new LubSwitch (t1) {
29     ...
30     public Ty caseVectorTy (VectorTy t1)
31     {
32       return new LubSwitch (t2) {
33         public Ty caseVectorTy (VectorTy t2)
34         {
35           SimpleTy ty = lub (t1.getElemTy (), t2.getElemTy ());
36           int len = Math.min (t1.getLen (), t2.getLen ());
37           return vectorTy (len, ty);
38         }
39       }.get ();
40     } ...
```

source: prpro.java/src/main/java/dsldesign/prpro/java/Types.java

Figure 6.10: *A Java implementation of join for simple and composite types in* `prpro`*; Only the case corresponding to* SVect *is shown*

representing a failure or success (Line 12). The result type is defined using the standard library type `Either` (Line 10). It captures either a successful result of joining as a type (a value of type `Ty`) or an error message.

Lines 14–23 correspond to the cases in Eq. (6.3); compare the code and the mathematical definition case-by-case. In Line 15, we delegate to the `lub` fuction for simple types. In lines 16–17, we apply the simple type join to element types, and the minimum function to the vector sizes. The cases for distrubution types follow Eq. (6.3) closely in the same fashion.

Figure 6.10 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 14–40, we show a fragment of lub for composite types. Like in Fig. 6.7, we use the switch pattern classes generated by the Ecore infrastructure. First, we specialize the switch class for the lub computation (16–25), with the most important part being the definition of the default case, producing an error message. Second, we instantiate it and show how to merge two vector types, with the core operations in Lines 35–37. The entire implementation is available from the book's code repository.

6.5 Type Checking Algorithm for `Prpro` 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 lowest positioned type containing the constant from the hierarchy of Fig. 6.5. 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 probability. Any other number between zero and one is typed as positive probability. Literals below zero (`-3.14`) are typed as `Float`. This is summarized in the following definition:

$$\text{type-of}(c) = \begin{cases} \mathsf{Nat} & \text{if } c \text{ is a positive integer literal} \\ \mathsf{NonNegInt} & \text{if } c = 0 \\ \mathsf{Int} & \text{for other integer literals} \\ \mathsf{Prob} & \text{if } c = 0.0 \\ \mathsf{PosProb} & \text{for } c \in (0;1] \\ \mathsf{PosFloat} & \text{for floating-point literal } c > 1.0 \\ \mathsf{Float} & \text{for other literals} \end{cases} \tag{6.4}$$

We directly assigned types to literals and constants, because their meaning is fixed and independent of the context. In contrast, an expression referring to variables depends on properties of these variables for its type. 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.

The typing rules assign a type (t) to each `prpro` expression (e) in a typing environment (Γ). We will use the following ternary *typing judgement* to state this formally and concisely:

$$\Gamma \vdash e : t \tag{6.5}$$

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, types a variable reference. The premise checks what type has been assigned to the variable `name` in the environment Γ and simply returns that type. 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:

$$(\textsc{Const})\frac{c \text{ is a literal}}{\Gamma \vdash c : \text{type-of}(c)} \qquad\qquad (\textsc{Var-Ref})\frac{\Gamma(\texttt{name}) = t}{\Gamma \vdash \texttt{name} : t}$$

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:

$$(\textsc{BExpr})\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t = t_1 \sqcup t_2}{\Gamma \vdash e_1 \oplus e_2 : t}$$

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: Nat` and `42: Nat` then it allows us to conclude that `(1 - 42): Nat` (the result should be `Int`)
2. Since `0.6: PosProb` and `0.42: PosProb` then it allows us to conclude that `(0.6 + 0.42): PosProb` (the result should be `NonNegFloat` since probability values cannot be larger than `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 the $t_1 \sqcup t_2$ for the binary expression case.

Implementation. Figure 6.11 presents an implementation of the rules CONST, VAR-REF, and BEXPR of the type-checker for expressions in Scala (Lines 1–22). The signature of function `tyCheck` corresponds to the

```
 1 def tyCheck (tenv: TypingEnvironment, expr: Expression): Result[Ty] =
 2   expr match {
 3     case BExpr (left, operator, right) =>
 4       for {
 5         t1 <- tyCheck (tenv, left)
 6         t2 <- tyCheck (tenv, right)
 7         t  <- lub (t1,t2)
 8       } yield expr.setTy (t)

10     case CstI (n) if n > 0 => Right (expr.setTy (NatTy))
11     case CstI (0) => Right (expr.setTy (NonNegIntTy))
12     case CstI (_) => Right (expr.setTy (IntTy))
13     case CstF (0.0) => Right (expr.setTy (ProbTy))
14     case CstF (x) if x > 0.0 && x <= 1.0 =>
15       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       tenv.get (name)
21         .map (expr.setTy)
22         .toRight (s"Undeclared variable '${name}'")

24     case Normal (mu, sigma, oobserved) =>
25       for {
26         _ <- tyCheck (tenv, mu).ensure (
27           t => t.isSubTypeOf (FloatTy),
28           t => s"Need a sub-type FloatTy for 'mu' but got '$t'"
29         )
30         _ <- tyCheck (tenv, sigma).ensure (
31           t => t.isSubTypeOf (NonNegFloatTy),
32           t => s"Need a sub-type of NonNegFloatTy for 'sigma', got '$t'"
33         )
34         _ <- oobserved
35           .map { observed =>
36             tyCheck (tenv, observed)
37             .ensure (
38               tob => tob.isSubTypeOf (VectorTy (1, FloatTy)),
39               tob => s"Need a vector of Floats for observed, got '$tob'"
40             )
41           }.getOrElse (Right (VectorTy (1, FloatTy)))
42       } yield expr.setTy (DistribTy (FloatTy)) ...
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/TypeChecker.scala

Figure 6.11: *The type checking rules for simple expressions (lines 3–22) of* `prpro` *and for a normal distribution node (lines 24–42) implemented in Scala*

structure of the typing judgement in the rules: it relates a typing environment (`tyenv` is Γ), an expression (`expr`) and a resulting type (`Result[Ty]`). Recall that `Result` is a type that allows to represent a type value inferred for an expression or a failure, including 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 5–6 or the `lub` in line 8 fail, then the entire `for-yield` fails and returns the error message.

```
 1 static public Ty tyCheck (Map<String, Ty> tenv, Expression expr)
 2   static class TyCheckExprSwitch extends TyCheckSwitch<Ty>
 3   {
 4     @Override public Ty caseCstI (CstI expr)
 5     {
 6       Ty result = Types.intTy;
 7       if (expr.getValue () > 0) result = Types.natTy;
 8       else if (expr.getValue () == 0) result = Types.nonNegIntTy;
 9       expr.setTy (result);
10       return result;
11     }

13     @Override public Ty caseBExpr (BExpr expr) throws TypeError
14     {
15       Ty t1 = tyCheck (tenv, expr.getLeft ());
16       Ty t2 = tyCheck (tenv, expr.getLeft ());
17       expr.setTy (Types.lub (t1, t2));
18       return expr.getTy ();
19     }

21     @Override public Ty caseNormal (Normal expr) throws TypeError
22     {
23       Ty t1 = tyCheck (tenv, expr.getMu ());
24       if (!Types.isSubTypeOf (t1, Types.floatTy))
25         throw new TypeError (
26          "Need a subtype of FloatTy  for ’mu’ but got ’" + t1 +"’" );

28       Ty t2 = tyCheck (tenv, expr.getSigma ());
29       if (!Types.isSubTypeOf (t2, Types.nonNegFloatTy))
30         throw new TypeError (
31          "Need a subtype of NonNegFloatTy for ’sigma’ but got ’"
32          + t2 +"’" );

34       if (expr.getObserved () != null) {
35         Ty tob = tyCheck (tenv, expr.getObserved ());
36         Ty stob = Types.vectorTy (1, Types.floatTy);
37         if (!Types.isSubTypeOf (tob, stob))
38           throw new TypeError (
39            "Need a vector of Floats for observed but got ’" + tob +"’" );
40       }

42       expr.setTy (Types.distribTy (Types.floatTy));
43       return expr.getTy ();
44     }
```

source: prpro.scala/src/main/scala/dsldesign/prpro/scala/adt/TypeChecker.scala

Figure 6.12: *Selected type checking rules for expressions of* `prpro` *implemented in Java, using the switch pattern with the infrastructure generated by Ecore*

Lines 10–17 implement the typing of literals based on CONST rule and Eq. (6.4). These cases cannot fail—a type has been defined for any literal. This is why they wrap the resulting type in the `Right` case of the `Result[Ty]`. (The `Left` case represents a failure.) Finally, lines 19–22 implement the VAR-REF rule. First, the typing environment, a `Map[String, Ty]`, is queried for the type of the variable referred with `name`. This results in a value of type `Option[Ty]`. If succeeded the `map` invocation will store this type in the annotation of the expression using a side effect for easy later access (Line 21). Function `setTy` has been declared in Fig. 6.1. Finally, the option value is converted with `toRight` to a result of the `Right[Ty]` if successful. If failed, `toRight` (slightly confusingly) creates a failing instance of `Left` encapsulating the provided error message.

Figure 6.12 presents the key part of the corresponding Java implementation. The function signature resembles the Scala type checker: we have a type environment (a map from names to types) and an expression to type. The function returns the inferred type directly. Instead of propagating errors using a special result type (in the functional style used in our Scala example), we opt for using exceptions here, as a more natural Java idiom. Consequently, a successfully inferred type is returned directly, while errors are propagated through the exception handling control flow. Otherwise the design is similar to Fig. 6.11. We use the switch pattern classes of Ecore again, providing an inner class to define visitors for various types, letting the Ecore generated machinery to handle the dispatching based on the types of traversed syntax nodes. The implementation of BEXPR is in lines 13–19, 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 (lines 4–11). VAR-REF is not included for brevity, but the entire implementation can be found in our source code repository.

Let us return to formal typing rules for `prpro`. The most domain specific part of `prpro` expressions are the constructors of probability distribution. Typing them is not very different from typing other expressions, only with slightly more idiosyncratic shape of sub-expressions and type requirements. 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 additionally provide a vector of data, which then can be used for probabilistic inference. From language design perspective (the only perspective relevant here), this means that the probability distribution expressions sometimes have an additional argument with data.

$$(\textsc{Norm})\frac{\Gamma \vdash e_\mu : \mathsf{Float} \qquad \Gamma \vdash e_\sigma : \mathsf{NonNegFloat}}{\Gamma \vdash \mathsf{Normal}(e_\mu, e_\sigma) : \mathsf{Distrib}(\mathsf{Float})}$$

$$(\textsc{Norm-O})\frac{\Gamma \vdash e_\mu : \mathsf{Float} \quad \Gamma \vdash e_\sigma : \mathsf{NonNegFloat} \quad \Gamma \vdash o : \mathsf{Vector}(1, \mathsf{Float})}{\Gamma \vdash \mathsf{Normal}(e_\mu, e_\sigma, o) : \mathsf{Distrib}(\mathsf{Float})}$$

$$(\textsc{Unif})\frac{\Gamma \vdash e_0 : t_0 \qquad \Gamma \vdash e_1 : t_1 \qquad t_0 \sqcup t_1 \leqslant t \quad t \text{ is simple}}{\Gamma \vdash \mathsf{Uniform}(e_0, e_1) : \mathsf{Distrib}(t)}$$

$$(\textsc{Unif-O})\frac{\Gamma \vdash e_0 : t_0 \quad \Gamma \vdash e_1 : t_1 \quad t_0 \sqcup t_1 \leqslant t \quad t \text{ simple} \quad \Gamma \vdash o : \mathsf{Vector}(1, t)}{\Gamma \vdash \mathsf{Uniform}(e_0, e_1, o) : \mathsf{Distrib}(t)}$$

The first variant of 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 second rule, NORM-O, adds an additional requirement that the observed property (o) gives a vector of floats, which basically means any vector of numbers here (why?). The last two rules follow the similar patter, but for uniform distributions. The interesting part 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 `NonNegFloat`. 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 `NonNegFloat`. We use the merging (lub) described in the previous section to find this type in the typing rules for uniform. Furthermore, when observed data (o) is provided (UNIF-O), we require that all the elements in this data set, also conform to this type.

Note that the UNIF-O rule is unsound: it is possible that the vector o contains values outside of the interval delimited by e_0 and e_1. This is unsound, since all such values are assigned probability zero, so either the uniform distribution is wrongly formulated, or the provided data set is inconsistent with it. For most applications this will be a problem. Unfortunately, this problem is hard to eliminate, especially if the endpoints of the interval are described by distributions themselves. In such case, we do not know their precise values at type checking time. Even for constant end-points of the intervals, we cannot enforce this rule without type checking data simultaneously with the model, which would require extending the language to provide some access path to the data set at the very least. In practice, such extensions to the type system, that reach out of the model to other connected data sources are very valuable, and help domain experts avoid errors when designing models. This is one aspect of domain-specific typing that is rarely seen in general-purpose programming languages, where the type systems tend to focus just on the program text, and ignore the external environment. We left this rules out of `prpro` for brevity reasons, though.

Finally, we need to include automatic up-casting in the typing rules (which allows to promote any type to its super-type):

$$(\textsc{Upcast})\frac{\Gamma \vdash e : t_1 \quad t_1 \leqslant t_2}{\Gamma \vdash e : t_2}$$

Practically, the Upcast rule allows using a distribution instead of a floating point number for parameters of normal and uniform distributions. The types t, NonNegFloat, and Float in probability rules are simple, but UPCAST will allow to promote a distribution type to simple types exploiting the sub-typing in SDIST-2 on page 217. This will allow to complete type checking for distribution expressions even if they nest other distribution expressions.

Implementation. Our implementations of type checking of probabilistic expressions have been shown in Fig. 6.11 (lines 24–42) and in Fig. 6.12 (lines 21–44). 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 if 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. Note that instead of separating NORM from NORM-O, like in our formal rules, we simply include the latter in the former conditionally (lines 33-40 and 34–40 respectively). This avoids some code repetition.

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 (substitability). 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 be actually returned for your example term from Exercise 6.4?

Finally, in our implementation there is nowhere a 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 27 of Fig. 6.11 and Line 24 of Fig. 6.12.

```scala
def tyCheck (tenv: TypingEnvironment, decl: Declaration)
  : Result[TypingEnvironment] =
    decl match {

      case Let (name, value) =>
        tyCheck (tenv, value)
          .ensure (
            t1 => tenv.get (name).isEmpty,
            t1 => s"'$name' has already been defined!" )
          .map { t1 => tenv + (name -> t1) }

      case Data (name, ty) =>
        tenv.get (name) match {
          case Some (_) =>
            Left (s"Identifier '$name' has already been defined!")

          case None =>
            Right (tenv + (name -> ty))
        }
    }
```

Figure 6.13: *The type checking rules for declarations implemented in Scala. This function has to be put in a loop iterating over the entire model to complete the type checker.*

6.6 Type Checking `Prpro` 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` declaration 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, LET updates the type environment Γ_0 with the type of a new variable `name`. The rule first infers the type of the expression e bound to `name`. Then ensures that `name` has not been previously defined. Finally, it captures the update in Γ_1:

$$(\textsc{Let})\frac{\Gamma_0 \vdash e : t \quad \Gamma_0(\texttt{name}) \text{ is undefined} \quad \Gamma_1 = \Gamma_0[\texttt{name} \mapsto t]}{\Gamma_0 \vdash \texttt{let name} = e : \Gamma_1}$$

$$(\textsc{Data})\frac{\Gamma_0(\texttt{name}) \text{ is undefined} \qquad \Gamma_1 = \Gamma_0[\texttt{name} \mapsto \texttt{t}]}{\Gamma_0 \vdash \texttt{data name of type t} : \Gamma_1}$$

$$(\textsc{Model})\frac{\Gamma_0 \vdash d_1 : \Gamma_1 \quad \cdots \quad \Gamma_{n-1} \vdash d_n : \Gamma_n}{\Gamma_0 \vdash d_1, \cdots, d_n : \Gamma_n} d_i \text{ are declarations}$$

The DATA rule, processing data set declarations, resembles LET, but, instead of inferring the type from an expression, uses the type specified directly in syntax. Finally, type checking the entire model requires that all declarations type check correctly, accumulating the types and names on the way (MODEL). A `prpro` model is *well-typed* if we can use the above rules to type all the declarations according to the last rule.

```java
 1  @Override public Map<String,Ty> caseLet (Let let) throws TypeError
 2  {
 3    String name = let.getName ();
 4    Ty t1 = tyCheck (tenv, let.getValue ());
 5    if (tenv.containsKey (name))
 6      throw new TypeError
 7        ("Identifier '" + name + "' has already been defined!");
 8    tenv.put (name, t1);
 9    return tenv;
10  }

12  @Override public Map<String,Ty> caseData (Data decl) throws TypeError
13  {
14    String name = decl.getName ();
15    if (tenv.containsKey (name))
16      throw new TypeError
17        ("Identifier '" + name + "' has already been defined!");
18    tenv.put (name, decl.getTy ());
19    return tenv;
20  }
```

Figure 6.14: *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.*

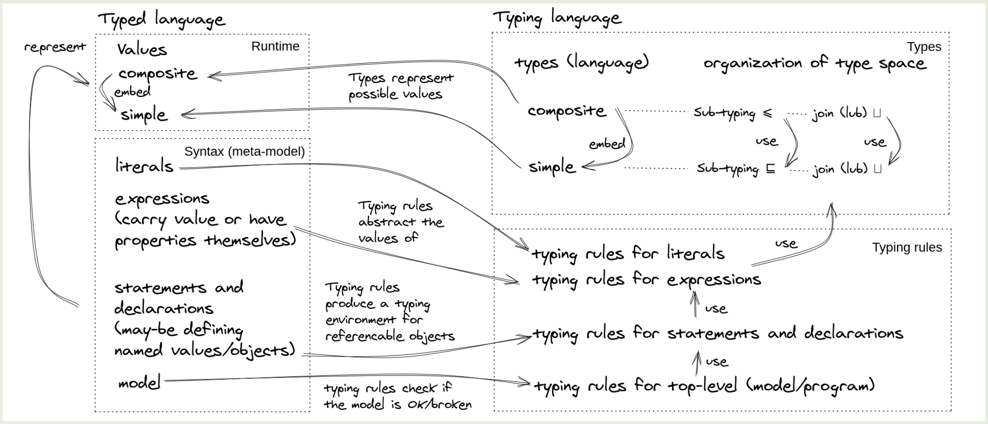
Implementation. Like before, the implementation of the above rules should ensure determinism. Rule LET shall not be satisfied with any type compatible with e, but should obtain the smallest, the most precise compatible type according to the sub-typing ordering. This makes it easier to reuse variables and expressions. If someone needs a Float and you give her a Prob, the model will still make sense, but not if you give a Float when 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.

The implementations of LET and DATA are shown in figures 6.13 (Scala) and 6.14 (Java). The former shows the entire implementation, while the latter only the overridden methods in the switch class for typing declarations. The LET rule first obtains the type of the right-hand-side expression. If successful, Line 8 (respectively 5) confirms that the variable had not been defined before. Both a failure of typing the expression and a repeated declaration of the same name cause the typing to stop with an error. Finally, the type environment is extended with a new mapping and returned (lines 5–10 and 1–10). The DATA rule, in both examples, just checks for repeated declaration, and if no problem is found, records the declared type in the typing map.

The MODEL rule (not shown for brevity reasons) is implemented using either a loop (imperative style) or a fold (functional style). It processes the declarations one-by-one using the above defined functions to build the typing environment. See the code repository for details.

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 ⊑
use
join (lub) ⊔
use
join (lub) ⊔
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 (**??**) 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 $\Gamma(\mathrm{x}) =$ 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 referencable properties that needs to be accessed later.

`let x=1` ensures that $\Gamma(\mathrm{x}) =$ Nat if $\Gamma(\mathrm{x})$ 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.

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.

Scenario-driven testing. Following the unit-test style, 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 if the implementation types it as prescribed. The second column in Table 6.1 shows examples of such test cases for `prpro`. For example, the first row has a test case derived from Fig. 6.5 to check whether the sub-typing implementation for simple types behaves as expected. The right column

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	`VectorTy (42, PosProbTy)`
	`  .isSubTypeOf (VectorTy(13,PosFloatTy))`
	`    must be (true)`
		`VectorTy (13, PosProbTy)`
		`  .isSubTypeOf (VectorTy (42, PosFloatTy))`
		`    must bet (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)`
		`lub (vectorTy (10, intTy),`
		`     DistribTy (probTy))`
		`  must fail`

type-of(t) for simple types	`typeOf(CstF (0.6)) must be (PosProbTy)`	no negative test (all literals are assigned a type)

VAR-REF	`VarRef ("x") with Map ("x" -> PosProbTy)`
	`  is of type (PosProbTy)`
		`VarRef ("x") with Map ("y" -> PosProbTy)`
		`  fails to type check`

NORM	`val e = Normal (CstF (0.42), CstF (0.1))`
	`tyCheck (tenv0 (), e) must be (FloatTy)`
		`val e = Normal (CstF (0.42), CstF (-0.42))`
		`tyCheck (tenv0 (),e) must fail`

LET	`Let("x", BExpr (CstI (1), Plus,`
	`  Normal(CstF(0.0),CstF(0.1))) with Map()`
	`  must be (Map ("x" -> Distrib(FloatTy)))`
		`Let ("x", BExpr (CstI (1), Plus,`
		`  Normal (CstF (0.0), CstF(0.1)))`
		`  with (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

shows negative test-cases, so examples of broken inputs to the type checker that violate the premises of typing rule. For the sub-typing of simple types in the first row, we choose two types that are not sub-types in Fig. 6.5.

There are typically many 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 well, is largely testing how it behaves on broken inputs.* In the SVECT row of the table, we show a violation of the sub-typing rule for vectors. The sub-type is shorter (13) than the super-type (42). Confirm on page 233 that this is indeed a negative test case. However, this is not the only one way to violate SVECT. For example, two types of the same lengths but incomparable element types, would fail 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, typing expressions (3 rows). The test cases for the final rule MODEL are not shown for brevity. 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 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 Table 6.1: SDIST-1, SDIST-2, BEXPR, NORM-O, 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 could establish general laws:

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 $t_1 \leq t_2$ then $t_2 \not\leq t_1$

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 main stream programming languages (e.g. Junit-quickcheck for Java,[5] hypothesis[6] 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 a lot (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

[5]https://github.com/pholser/junit-quickcheck
[6]https://github.com/HypothesisWorks/hypothesis

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) =>
  inside (lub (t1,t2)) {
    case Right(ty) =>
      t1.isSubTypeOf (ty) must be (true)
      t2.isSubTypeOf (ty) must be (true)
    case Left(msg) => }}
```

join is the least super-type of arguments

```
forAll { (t: Ty, t1: Ty, t2: Ty) =>
  whenever (t1.isSubTypeOf (t) && t2.isSubTypeOf (t)) {
    inside (lub (t1,t2)) {
      case Right (ty) =>
        ty.isSubTypeOf (t) must be (true)
      case Left (_) =>
        fail ("must exist if a super-type does!") }}}}
```

source: prpro.scala/src/test/scala/dsldesign/prpro/scala/adt/TypesSpec.scala

Table 6.2: Examples of property tests for `prpro` type system. Note that these tests are the same for any type system. Code examples use Scalatest library, but similar Quickcheck-style libraries exist for any main stream programming language

```scala
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 6.15: *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*

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.

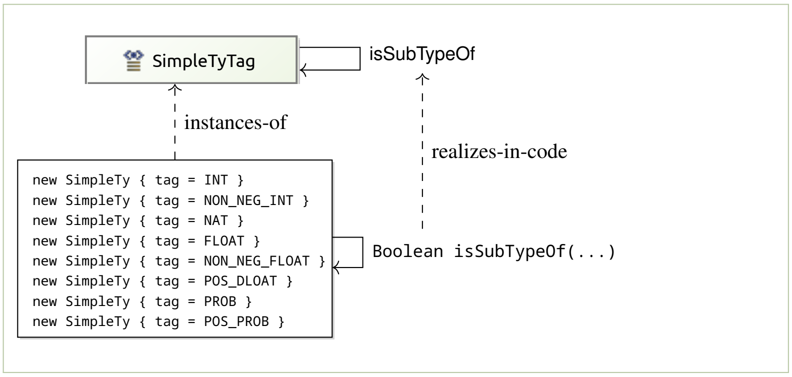
Property-based testing needs a way to generate inputs automatically. For all the types tested by properties, one needs to create generators compatible with the framework. Figure 6.15 shows the generator for our composite types. It is a inductive procedure that constructs larger terms from smaller terms, 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 (**??**).

Exercise 6.8. Implement properties for testing the two laws stated on page 234 and use them to test our type system.

Exercise 6.9. Write a regression scenario test exposing the unsoundness of the 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 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*. 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. The *shallow embedding* is the dual pattern, popular in code generators and in internal DSLs. We define and discuss it in **????**.



```
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 6.16: *The subtyping hierarchy is typically not implemented using subtyping in the implementation language (a shallow embedding) but it is coded separately as a function related values (a deep embedding).*

Definition 6.6 (Deep embedding)**.** *Deep embedding is a language implementation pattern in which the elements of the implemented language (say types) are represented as values in the implementation language, and not using the corresponding elements of the implementation language.*

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 6.16 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. 6.5 is not captured by Java sub-typing (inheritance) but it becomes an association between values. We have implemented this association not as a direct reference but as a function `isSubTypeOf` that derives the property from all super types of each type (Fig. 6.7 line 10, and Fig. 6.6 line 3).

Figure 6.17 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. 6.2). 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. 6.17, an exact copy of Fig. 6.4. Again, the vertical dashed lines mark the instantiation relations.

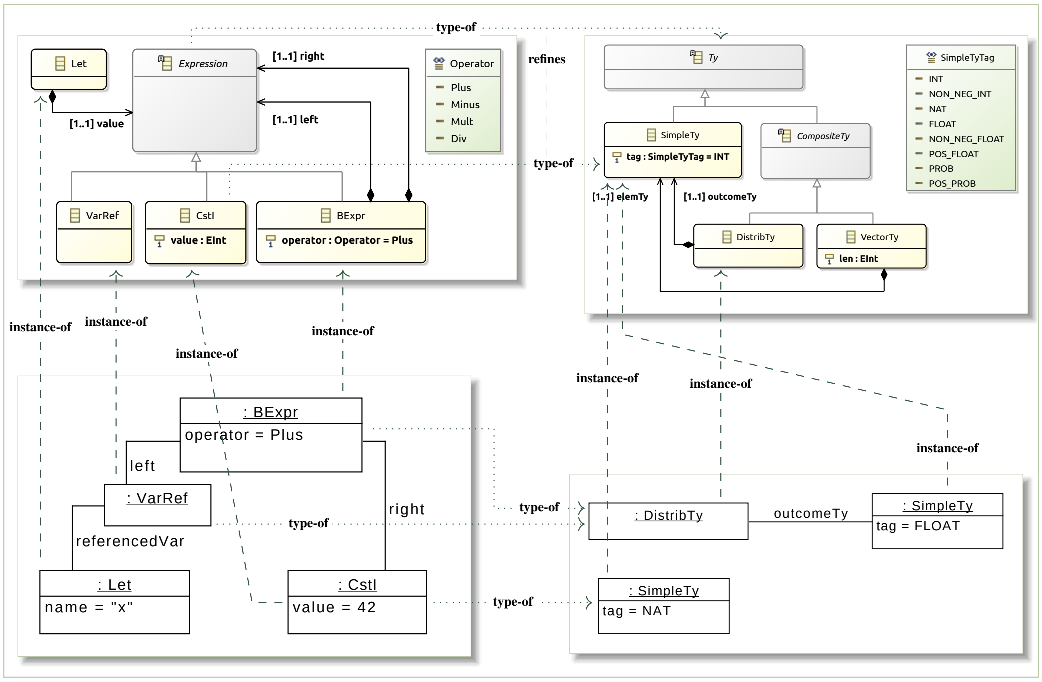


Figure 6.17: Relating the typed language (top left), the typing language (top right), the typed language instance (bottom left), and the actual typing (bottom right). Dashed arrows represent instantiation (conformance), while dotted arrows represent typing

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`. Note that at the meta-level no concrete types can be 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 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 good 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. For instance, 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 (soundness)? How to design type systems that do not require explicit type annotations and support generic functions (parametric polymorphism) with true type inference (solving type variable constraints)? The classic introductory text on type systems that goes into considerable detail is the book of Pierce (2002). The goal of Lämmel (2018) 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 (2012).

The ultimate goal of many researchers in type system engineering is to depart from manual implementation of type checkers, to generate them automatically from high-level descriptions, in a way similar to how we generate parsers from grammars. Antwerpen et al. (2016) and Pelsmaeker, Antwerpen, and Visser (2019) describe a tool, Statix,[7] 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 being developed within the ecosystem of the Spoofax[8] language workbench (Kats and Visser, 2010). Another tool for declarative definitions of type systems (and interpreters), integrated with Xtext, is Xsemantics[9] (Bettini, 2013).

In `prpro`, we have developed an entire expression language for the purpose of the example. Many DSLs use a similar generic expression language. Xbase[10] 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 that originated with the seminal paper of Claessen and Hughes (2000) 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 (Palka et al., 2011). Generating meaningful random models and programs remains a challenge, for instance ensuring that enough of them are well-typed (Midtgaard and Møller, 2017).

Additional Exercises

Exercise 6.10. Add vector literal to the abstract syntax of `prpro` (the possibility of writing constant literals that are representing vectors). Why the typing hierarchy does not need to be changed with this extension? Expand the typing rules to account for the new construct, and implement the new rule.

Exercise 6.11. Extend `prpro` with type-casting, so the ability to force a type of an expression. For instance (in Scala notation) we could imagine a constructor:

```
case class Cast (e: expression, ty: Ty) extends Expression
```

[7] https://eelcovisser.org/research/#Statix
[8] http://www.metaborg.org/en/latest/
[9] https://github.com/eclipse/xsemantics
[10] https://www.eclipse.org/Xtext/documentation/305_xbase.html

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.

Notice that this extension provides another workaround for the weaknesses of the numeric type inference in `prpro`. If the type system is not able to prove that a number is positive, the programmer may explicitly indicate it. Of course, if the programmer is wrong, the model will be malformed. It would be prudent to insert a runtime check at this point in the interpreter for `prpro`, to ensure that the type cast is safe for the value produced by *e*. We will revisit this question in **??**, **??**.

Exercise 6.12. Add 2-dimensional vector types to the type language of `prpro` (so 2-dimensional arrays, or vectors of vectors) and the typing rule (next to SVECT) to support it.

Exercise 6.13. (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 6.14. (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 (Figs. 6.1 and 6.2). 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. 6.13) for `prpro` and test that it works well.

Exercise 6.15. (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 per each value of the mean. (A "vectorized" distribtuion 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 6.16. (A small project) The choice that distributions are subtypes of simple numbers (see rule SDIST-2) may seemed arbitrary. For instance, it allows vectors of distributions through a back door. Alternatively, we could invent an abstract type, say `ArithmeticValueTy`, linked to the interface of objects that participate in arithmetic expressions. Both simple types and distribution types could become its subtypes. Redefine the typing language to change the design in this direction. This requires changing the typing rules for arithmetic expressions (to take value types). Reflect how would we would now make vectors to be arithmetic values as well.

Exercise 6.17. (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 6.18. (A small project) Revisit the finite state machine language from Chapter 3. Add global numeric variables and expressions (both arithmetic and Boolean) to the abstract syntax, and allow adding Boolean expressions as guards on transitions. A transition is active and can be taken if the source state is active and the guard condition evaluates to true. Finally, design a type system that ensures that only Boolean expressions, not numeric expressions, are used as guards on transitions.

Exercise 6.19. (A project) Design a simple language for electric circuits. Each wire in a circuit can carry a DC power or AC power. We have power inputs, and power outputs. We have wires that have two ends (in and out) that can be connected to other wires or junctions. Junctions that connect several incoming wires to outgoing wires. Finally, we have a frequency converter (one input, one output) that can change incoming AC current into a DC current. The abstract syntax can be designed as a simplification of the finite state machine language.

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

Antwerpen, Hendrik van et al. (2016). "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. DOI: 10.1145/2847538.2847543. URL: https://doi.org/10.1145/2847538.2847543.

Bettini, Lorenzo (2013). "Implementing Java-like languages in Xtext with Xsemantics". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, pp. 1559–1564.

Claessen, Koen and John Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by Martin Odersky and Philip Wadler. ACM, pp. 268–279. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, pp. 237–238. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869592. URL: http://doi.acm.org/10.1145/1869542.1869592.

Lämmel, Ralf (2018). *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer.

Liskov, Barbara H. and Jeannette M. Wing (Nov. 1994). "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6, pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

Midtgaard, Jan and Anders Møller (2017). “QuickChecking static analysis properties”. In: *Softw. Test. Verification Reliab.* 27.6. DOI: 10.1002/stvr.1640. URL: https://doi.org/10.1002/stvr.1640.

Palka, Michal H. et al. (2011). “Testing an optimising compiler by generating random lambda terms”. In: *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*. Ed. by Antonia Bertolino, Howard Foster, and J. Jenny Li. ACM, pp. 91–97. DOI: 10.1145/1982595.1982615. URL: https://doi.org/10.1145/1982595.1982615.

Pelsmaeker, Daniël A. A., Hendrik van Antwerpen, and Eelco Visser (2019). “Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications”. In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.26. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.26.

Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.

Sestoft, Peter (2012). *Programming language concepts*. Springer Science & Business Media.

7 Design Patterns and Practices for Concrete Syntax

7.1 Placeholder

This chapter is just a placeholder for now, so that we can refer to it.

8 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 the modeling aspect when introducing SPLE and its main concepts. 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 languages 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 *feature* in the remainder) can be present or absent in a concrete variant of the product line. But, 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. 8.2), while for the latter, dedicated DSLs need to be created (see our case study of fire alarm systems in Sect. 8.8).

8.1 The Need for 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.

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 using the language *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 both.

Creating variants of software systems allows organizations to address such varying stakeholder requirements. It allows them to experiment with new 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 had already been developed in an existing project. This allows her to reuse past effort very easily and very 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 both 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 is growing. If this continues, the costs are growing with the age of the projects, and ultimately the entire system family becomes too expensive to maintain.

Similar problems appear when this so-called *clone & own* (Dubinsky et al., 2013; Businge et al., 2018; Stanciulescu, Schulze, and Wąsowski, 2015) 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 (McKee et al., 2017; Menezes, 2016; Mahmood et al., 2020; Accioly, Borba, and Cavalcanti, 2018), 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 the one by Staples and Hill (2004) can help. But, in general, version control is not well-suited to organize many variants of software in parallel over time. It is much better suited to organize sequential variants—a.k.a., history, representing software evolution in time. Branches and forks should be used to organize the development process (for instance, using feature branches) and not the architecture.

Opportunistic software reuse in terms of clone & own is in fact the most common strategy that organizations use for creating software variants (Berger, Rublack, et al., 2013). Various companies have documented their experiences (Fogdal et al., 2016; Staples and Hill, 2004; Duc et al., 2014; Dubinsky et al., 2013; Berger, Steghöfer, et al., 2019) of using clone & own. There are also many open-source projects handling their variants with this strategy, such as open-source software and firmware (Stanciulescu, Schulze, and Wąsowski, 2015; Krüger et al., 2019), families of Android apps (Businge et al., 2018; Mojica et al., 2014), families of Java and Android games (Akesson et al., 2019; Debbiche et al., 2019; J. Krüger et al., 2018), web applications (Ji et al., 2015), as well as robotics control software (Garcia, Strueber, Brugali, Fava, et al., 2019; Garcia, Strueber, Brugali, Berger, et al., 2020).

Let us take a look at the 3D printer firmware Marlin (Stanciulescu, Schulze, and Wąsowski, 2015; Krüger et al., 2019; J. Krueger, Gu, et al., 2018), which has over 8000 forks nowadays. Almost 20 % of these forks represent different variants (Stanciulescu, Schulze, and Wąsowski, 2015), since new features were developed within them, for instance, to support new printer models. Interestingly, many other forks just change the configuration file of Marlin as a very pragmatic way of storing individual configurations.

Forking provides substantial flexibility and drives innovation in Marlin (Stanciulescu, Schulze, and Wąsowski, 2015). It allows to experiment, 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 (Gousios, Pinzger, and Deursen, 2014). Practically, 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.

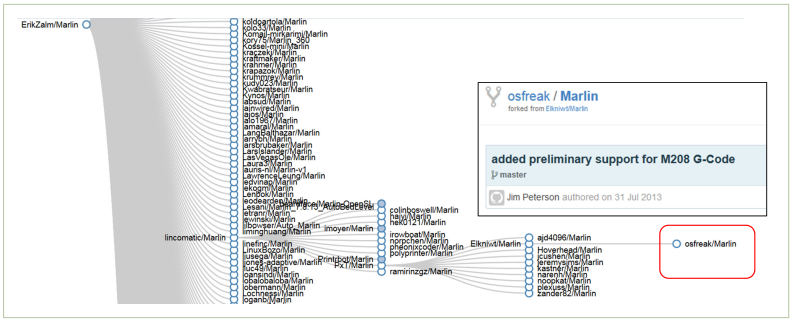


Figure 8.1: *Features hidden in the Marlin fork ecosystem (image courtesy of Ștefan Stănciulescu)*

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 (Stanciulescu, Schulze, and Wąsowski, 2015). 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 loose overview of the forks and their content. Finding interesting additions and features becomes challenging. Figure 8.1 shows that sometimes, new features can be hidden in the fourth level of forks from the main Marlin project. So, developers can easily loose overview over the features that exist in the fork ecosystem, as well as they might not be aware 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 a software that 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 to 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 those 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 (Berger, Pfeiffer, et al., 2014): 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 (Bosch, 2009; Jansen, Finkelstein, and Brinkkemper, 2009; Berger, Pfeiffer, et al., 2014). 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 super-computer 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 ecosystem uses different variability mechanisms and strategies to systematically manage variability, as illustrated in Fig. 8.2. Linux and eCos use feature models, which are hierarchical menus of configuration options and their dependencies (explained in detail in Sect. 8.5.2). 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. 8.2, 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

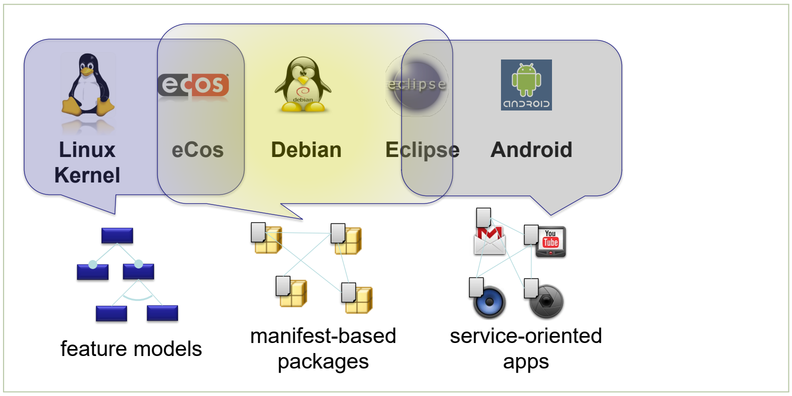


Figure 8.2: *Variability mechanisms in five software ecosystems*

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 this mechanisms, we refer to Berger, Pfeiffer, et al. (2014).

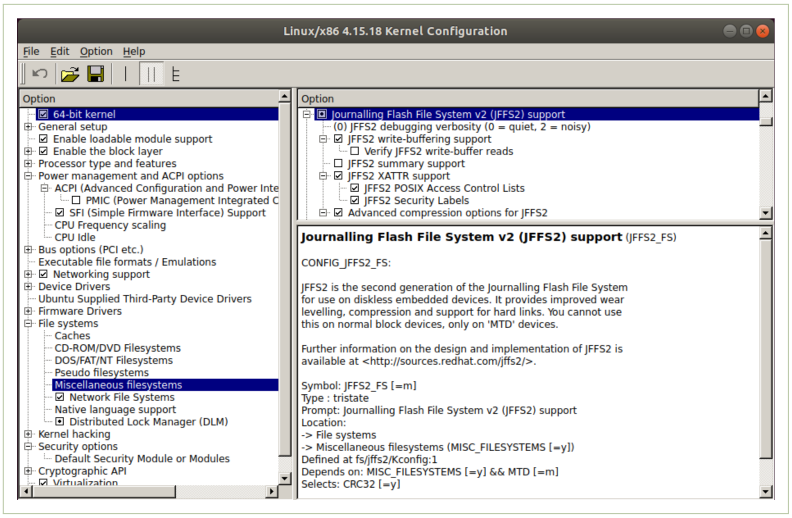
In the remainder, our focus is on feature models as a language[1] that is not only confined to systems such as the Linux kernel or other software product lines, but feature models can be seen as a very intuitive language to model systems, domains, concepts, or other languages.

8.2 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 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 into the Linux kernel. 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 Mecklenburg, 2004. 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 8.3 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,

[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 (K.C. Kang, 2009; Berger and Collet, 2019).



```
☑ 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
```

```
⊡ 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 8.3: *The Linux kernel configurator, showing the Kconfig language's concrete syntax*

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. 8.3). 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 (Berger, She, Lotufo, Krzysztof Czarnecki, et al., 2010) for compilation. Specifically, the build system selects the files relevant for the selected features—more precisely, the files whose presence condition evaluates to true[2]—and then the C preprocessor outputs C source files that are customized via 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 re-building 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 (Berger, She, Lotufo, Wąsowski, et al., 2013) 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 (Nadi et al., 2015), for instance, when the code included by one

[2]It is actually even more complicated, since the build system does not use explicit presence conditions explicitly, but they are encoded using some convention. See Berger, She, Lotufo, Krzysztof Czarnecki, et al. (2010) for more details.

feature references code that is only included by another feature. But, 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* (Zippel, 2017). The DSL has one graphical and multiple textual syntaxes, implemented in the respective configurator tools (Fig. 8.3 shows the graphical configurator). 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 is also used in various other systems software projects, such as Busybox and embedded libraries, such as uClibc (Berger, She, Lotufo, Wąsowski, et al., 2013).

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 by 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 (Lotufo et al., 2010). We can clearly see that the evolution of the kernel is feature-driven, since the code and the Kconfig model do co-evolve. When changing or adding features (e.g. a device driver), usually, developers 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 (Berger, She, Lotufo, Wąsowski, et al., 2013; She and Berger, 2010; El-Sharkawy, Krafczyk, and Schmid, 2015). Feature models are the most popular notation for modeling features and their constraints, and which we will discuss in detail in Sect. 8.5. 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 how far away they are in the hierarchy) can be expresses as so-called cross-tree constraints. To this end, Kconfig provides a simple constraint language with three-state logics (Kleene, 1938) for controlling the binding mode of features (a feature of

```
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 8.4: Kconfig excerpt for a filesystem (JFFS2) available in the Linux kernel, shown in textual concrete syntax

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, She, Lotufo, Wąsowski, et al. (2013). We will also explain feature modeling in more detail shortly, in Sect. 8.5.2.

Let us look at a small excerpt of the Linux kernel model that is shown in Fig. 8.4. 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. 8.7 (we explain feature models shortly in Sect. 8.5.2). 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.
- 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 logics (Kleene, 1938)): "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, but to 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 here, 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 relatively complex language 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 with Kconfig. 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 (e.g., via choice propagation).

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 could help users substantially when they need to enable or disable features, which requires enabling or disabling other features, and so on. 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 (Hubaux, Xiong, and Krzysztof Czarnecki, 2012) 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 was 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 challenges extending the configurator or building 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 (Russell and Norvig, 2016). Using such a reasoner, however, requires transforming the kernel model into the logical representation needed by the solver (e.g., a propositional logics formula in conjunctive normal form in the case of a SAT solver). This, in turn, requires understanding the exact syntax and semantics of Kconfig in order to develop a valid model transformation. Unfortunately, from our own experience, 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 (She and Berger, 2010; Berger, She, Lotufo, Wąsowski, et al., 2013). Formally defining syntax and semantics took over one month, and implementing the transformation into a propositional logics formula another few months. Various other researchers also implemented transformations themselves later, also being challenged by Kconfig's complexity, as shown in a survey by El-Sharkawy, Krafczyk, and Schmid (2015).

Another open-source DSL used in systems software is CDL (Component Definition Language) (Berger, She, Lotufo, Wąsowski, et al., 2013; Berger and She, 2010), 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 done to a configuration to allow setting a feature to a certain value.

Exercise 8.1. The Linux kernel and the Android operating system are two prominent examples of variability-rich systems. After we 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, you could read the paper by Berger, Pfeiffer, et al. (2014).

Note that there is no notion of completeness for your discussion, and apparently, you cannot cover all the details given in the paper. But, cover the aspects above (plus the two you find relevant). See it from the perspective that you tell 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 designs, for some software platform.

8.3 Software Product Line Engineering

Let us now look into 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 (Parnas, 1976).

SPLE arose from the observation that *opportunistic reuse does not scale with the number of software variants*, as we discussed above in Sect. 8.1. The following is a well-established definition of what a software product line is:

Definition 8.1. *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

[3]By L. M. Northrop (2010)

line represents a *portfolio of software products* ("set of software-intensive systems"). Note that we will use the terms variant, product, and system synonymously in the remainder. 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 (Apel, Batory, et al., 2013b; Krzysztof Czarnecki and U. W. Eisenecker, 2000).

Organizing your software production into a product line is usually linked with an intention of addressing a certain well-scoped market niche, by providing well-customizable software for this niche/target group. 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, then it is usually not meaningful to develop them as a product line. They likely do not share enough commonality that can be exploited for establishing a platform.

SPLE is a method in which technical, business, and management issues overlap. Adopting SPLE requires considering the four concerns Business, Architecture, Process, and Organization, which is called the BAPO model (F. J. v. d. Linden, Schmid, and Rommes, 2007; Obbink et al., 2000; F. v. d. Linden, 2002). 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. *Organisation* 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.

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 deriving the individual products. Both are full-blown classic engineering processes. Figure 8.5 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 these efficiently when building new systems. *Application engineering* (bottom) derives the artifacts from the common domain artifacts

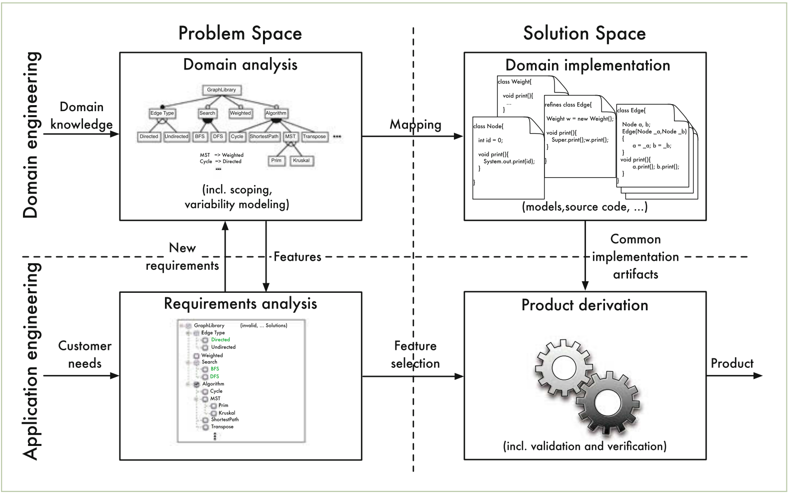


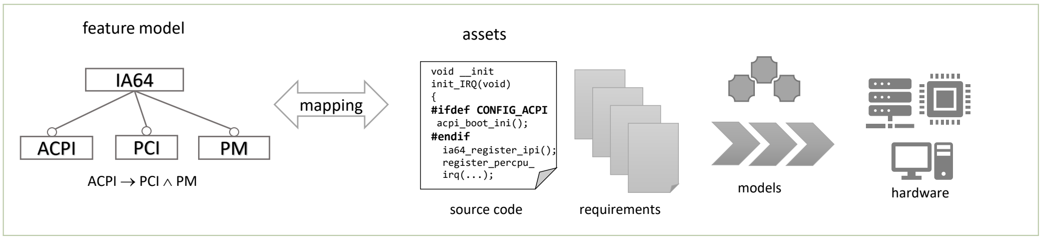
Figure 8.5: *The two main processes of SPLE:* domain engineering *and* application engineering. *Figure from Apel, Batory, et al. (2013b).*

produced in domain engineering (the top process). So, the design is done by completion of the shared design, 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 lowered. Observe that the vertical arrows in Fig. 8.5 (derivation of applications from platform assets) are obtained using technologies presented in the previous chapters.

Let us also briefly look into the concern *Architecture*, which is typically realizing two abstractions: the problem space and the solution space (Krzysztof Czarnecki and U. W. Eisenecker, 2000). Figure 8.5 and Fig. 8.6 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 (Krzysztof Czarnecki and U. W. Eisenecker, 2000; Apel, Batory, et al., 2013b). 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, we refer to the book by Apel, Batory, et al. (2013a), which describes many different ways to realize it using different implementation techniques.

There exists a mapping between problem and solution space, which can be realized using different techniques. An important concept is the presence condition.

Definition 8.2. *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.*



```
void __init
init_IRQ(void)
{
#ifdef CONFIG_ACPI
 acpi_boot_ini();
#endif
  ia64_register_ipi();
  register_percpu_
  irq(...);
```

Figure 8.6: *High-level architecture of a product line, illustrating the mapping of a feature model (problem space) to different types of assets (solution space)*

Looking at our running example, the Linux kernel, observe that it has presence conditions, which are contained in the preprocessor directives (e.g., `#if`) and, implicitly, in its build system (Berger, She, Lotufo, Krzysztof Czarnecki, et al., 2010). 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, we emphasize that 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 8.6 illustrates a system where features are mapped to code, requirements, models, and pieces of hardware, which is a typical set of asset types features are mapped to in industry.

8.4 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, an interactive configurator tool, and a DSL- and model-based representation of all kernel 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 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 (Berger, Nair, et al., 2014a) and Scania (Eklund and Håkan Gustavsson, 2013; Hakan Gustavsson and Eklund, 2010), Audi (Hardung, Kölzow, and A. Krüger, 2004), Daimler (Dziobek et al., 2008; Bayer, Forster, et al., 2006), General Motors (Flores, C. Krueger, and Paul Clements, 2012), Rolls-Royce (Habli and Kelly, 2007), as well as the engine control softwares of Bosch (Tischer et al., 2011) or Cummins (P. Clements and L. Northrop, 2001).

- Avionics and aerospace is another domain benefiting from SPLE, which in addition has strict safety requriements. Example experience reports have been written about Eurocopter (Dordowsky and Hipp, 2009; Hess and Dordowsky, 2008), Lufthansa (Chastek et al., 2011), NASA (Ganesan et al., 2009), Boeing (Sharp, 1998), and the US Army's Common Avionics Architecture System (CAAS) (Paul Clements and Bergey, 2005).
- Telecommunication is another typical domain suited for SPLE. Consider the experience reports about Ericsson (Svahnberg and Bosch, 1999; Mohagheghi and Conradi, 2008; Andersson and Bosch, 2005), E-COM (Liang, Hu, and Wang, 2005), Terrestrial Trunked Radio (TETRA) (Pohjalainen, 2011), as well as Nokia Mobile Phones and Nokia Networks (F. J. v. d. Linden, Schmid, and Rommes, 2007).
- Power electronics systems often need to exist in many different variants, as discussed in the experience reports about Danfoss (Fogdal et al., 2016), ABB (Ganz and Layes, 1998; Rösel, 1998; Pohl, Böckle, and F. J. v. d. Linden, 2005; Stoll et al., 2009), and Hitachi (Takebe et al., 2009).
- Robotics and industrial automation systems often come in different hardware configurations and benefit from SPLE methods, as discussed in a case study on re-engineering automation systems into product lines by Koziolek et al. (2016), in an experience report about managing variability in robotics (Garcia, Strueber, Brugali, Fava, et al., 2019), in one of the case studies on the company Keba in Berger, Lettner, et al. (2015), and one of the companies in Berger, Nair, et al. (2014a).
- Even web applications have been reported (Verlage and Kiesgen, 2005). While specific architectures for web applications have been proposed (Balzerani et al., 2005), they often exhibit variability in the user interface, which is still difficult to implement (Berger, Steghöfer, et al., 2019).
- Further case-study collections are provided by F. J. v. d. Linden, Schmid, and Rommes (2007), the SEI's catalog of case studies Software Engineering Institute (n.d.), and the SPLE community's "Hall of Fame."[4] All are summarized in Berger, Steghöfer, et al. (2019).

8.5 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 case of the Linux kernel) that

[4] http://www.splc.net/fame.html

describes the differences and similarities between systems, and then link this model to the implementation either via code generation (generating individual products) or 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 the 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 8.3. *A* variability model *is a domain model that describes the common and variable aspects of software products in a software product line.*

For software product lines, or for any complex system, it is not sufficient to only model the variability—that is, how the individual software products differ. To support the engineering (e.g., to keep an overview understanding or to scope a future 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.

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" (Krzysztof Czarnecki and U. Eisenecker, 2000).
- *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 constrol systems. An example of a product family in a horizontal domain is the Linux kernel,[5] or a configurable platform for cloud computing.

The scope of the domain defines how diverse products will be allowed 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 of 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 of the domain, defines its vocabulary and the main concepts of the domain. Domain models can be expressed in many ways, but most commonly as a DSL. We call these DSLs variability modeling languages.

Variability Modeling Languages

Since the advent of SPLE to efficiently develop software product lines in the early 1990s, a large number of variability modeling languages has been proposed (Sinnema and Deelstra, 2007; Lianping Chen, Muhammad Ali Babar, and Ali, 2009; Alves et al., 2010; L. Chen and M. Ali Babar, 2009). Variability modeling is one of the primary means to tackle the complexity of product lines, describing the variability and the commonality of all the variants that belong to a product line.

The most popular languages are feature (Kyo Kang et al., 1990; K.C. Kang, 2009) and decision models (Schmid, Rabiser, and Grünbacher, 2011a), which are relatively similar with only smaller differences K. Czarnecki et al., 2012. 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 (Berger, Rublack, et al., 2013). 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 (*Reuse-Driven Software Processes Guidebook, Version 02.00.03* 1993). Schmid, Rabiser, and Grünbacher

[5] 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.

(2011b), Dhungana, Heymans, and Rabiser (2010), and K. Czarnecki et al. (2012) provide further information about decision models. An alternative language is OVM (Orthogonal Variability Modeling), which focuses on modeling variation points and its variants Pohl, Böckle, and F. v. d. Linden (2005). Surveys comparing variability modeling languages from different perspectives are provided by Classen, Heymans, and Pierre-Yves Schobbens (2008), P.-Y. Schobbens et al. (2006), Schmid, Rabiser, and Grünbacher (2011b), K. Czarnecki et al. (2012), and Sinnema and Deelstra (2007).

An interesting alternative is to provide a variability modeling language as a UML profile (Ziadi, Hélouët, and Jézéquel, 2004), allowing to add variation points to ordinary UML diagrams, such as class diagrams or state machines. It relies on UML's built-in extension mechanism called profiles (Object Management Group, 2017). 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 diagram variation points and their mappings to features.

Feature Modeling

An important notation for expressing domain models is *feature models*. Feature models are a simple, tree-based modeling notation that allows expressing features and their constraints. 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 (Kang et al., 1990), hundreds of variability management methods and tools have been introduced that build upon feature models. Nowadays, many different variants of the original feature-modeling notation exist. A brief history of these notations is provided by Berger and Collet (2019).

Feature models center around the notion of feature (Berger, Lettner, et al., 2015). Features are abstract entities used in a multitude of contexts, including software configuration, product marketing, scoping, requirements engineering or 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 (Passos et al., 2018).

The notion of features is vague, which is in fact one of their core strengths, since organizations can choose their own definition. In the most general sense, we can say that features abstractly represent functional or non-

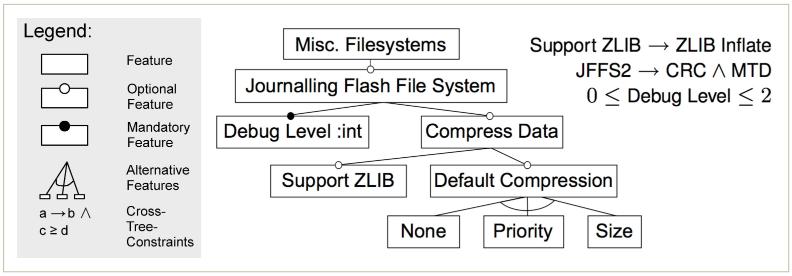


Figure 8.7: *Configuration of a Linux filesystem illustrated as a feature model (concrete syntax)*

functional concerns of a software system. We can also see features as end-user-visible characteristics of a system (Kang et al., 1990; Berger, Lettner, et al., 2015), 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 (Bosch, 2000), others only the business side (Riebisch, 2003). 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 which can provide the basis for a more concrete definition that an organization can formulate.

Definition 8.4. *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.*

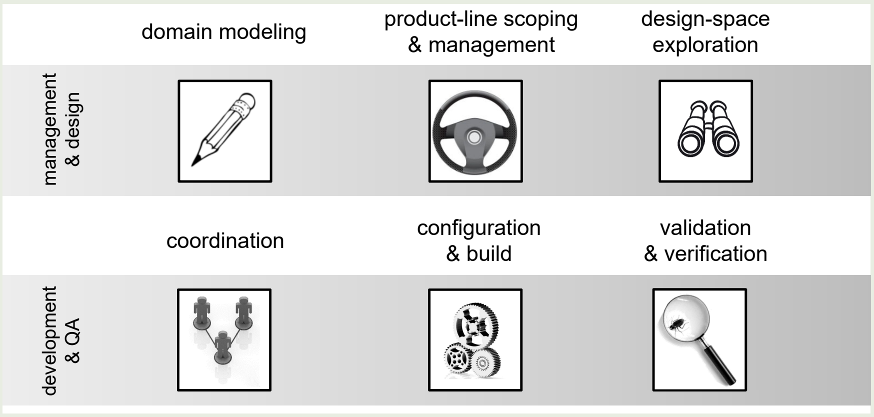
Feature models organize features in a hierarchy as well as they declare relationships and constraints among features. Feature models allow developers keeping an overview understanding of software systems, and like features, are an intuitive means for communications, bridging different kinds of stakeholders, including developers and domain or business experts.

Figure 8.7 shows a feature model example. Take a look at the legend, which explains the basic syntax. *Mandatory* features (filled circle) are features that 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).

The example model in Fig. 8.7 is a feature model we created for a configurable file system in the Linux kernel called JFFSs (Journalling Flash

Uses of Feature Modeling

As illustrated below, feature models are used for different purposes. We distinguish between management & design uses, such as for domain modeling, scoping and managing the product line, as well as performing design-space exploration, and between 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 (Beuche, 2004), Gears (C. W. Krueger, 2007) from the company BigLever, or the open-source tool FeatureIDE (Thüm et al., 2014). 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.

File System). In reality, it is defined in the Kconfig language, which we explained above in Sect. 8.2, and 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 to select exactly one sub-feature.

Additional dependencies between features (those that cross the tree hierarchy) can be stated on the side. In our example, we show three such constraints, which are typically called cross-tree constraints (CTCs). Note that ZLIB Inflate is a feature that is defined outside our excerpt of the Linux kernel model.

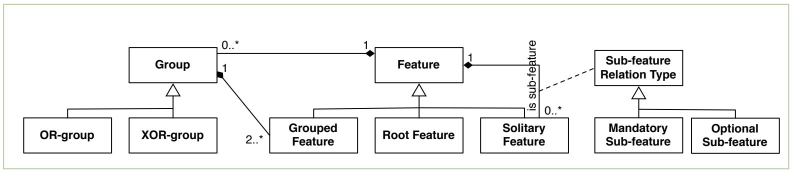


Figure 8.8: *One possible meta-model for feature models (Janota, Kuzina, and Wąsowski, 2008)*

Definition 8.5. *A* feature model *is a tree-based structure representing features and their constraints.*

For completeness, we include a possible meta-model for feature models in Fig. 8.8. 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).

After discussing the syntax and typical uses of feature models, let us now look into their semantics. 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 is to represent 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.

The products or variants are represented as a configuration of the feature model.

Definition 8.6. *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. When features are mapped to variation points, then they control the inclusion of certain assets depending on a concrete configuration of a feature model. Here, also recall Fig. 8.6 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. 8.2 above.

Definition 8.7. 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.*

8.6 The Process of Feature Modeling

Now that we discussed the feature modeling notation in detail, and also sneaked into the feature-modeling-like language Kconfig used in the Linux

Symbol	Description
?	Decision affecting following activities
⚙	Activity
(⚙)	Optional activity
⚙⚙	Composite activity
(⚙⚙)	Optional composite activity
⚙	Sub-Activity of a composite activity

Table 8.1: *Legend for feature modeling guidelines*

kernel and other systems software projects (Berger, She, Lotufo, Wąsowski, et al., 2013), let us look into the modeling process itself. In this section, we will refer to our modeling principles presented in Nesic et al. (2019) and present core questions you need to answer as well as different kinds of actions you need to perform. Table 8.1 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 (C. Krueger, 2002; Berger, Rublack, et al., 2013), 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 with 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*, then 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 M5: 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 271.

In our process, we classify the different activities into four phases: Pre-Modeling, Domain Analysis and Scoping, Modeling, and Maintenance and Evolution. Figure 8.9 depicts these phases, together with typical iterations among the last three phases.

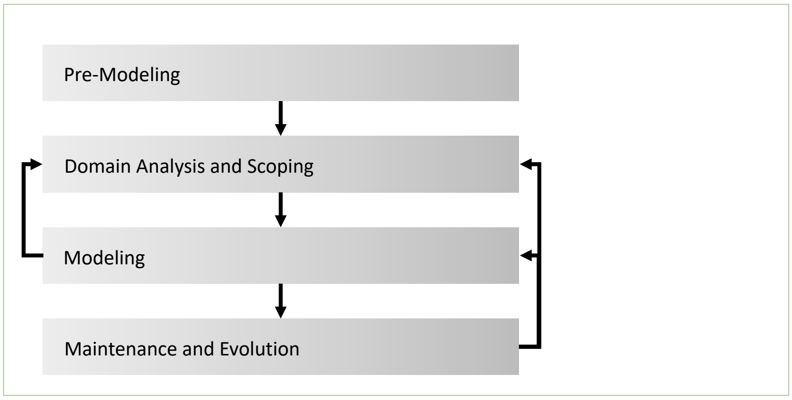


Figure 8.9: *The four main phases of the feature modeling process*

Pre-Modeling Activities

In the first phase, before you start with 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 defining the model purpose (activity Define Model Purpose) and training (activity Provide Training) in iteration, which allows clarifying and refining the purpose.

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), not wasting time with irrelevant ones. Choose among the different uses shown in the infobox on page ??. 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 towards the business, while at the same time it should be possible to map the features towards the software assets and control the platform, that is, be able to derive individual products in an automated process supported by a configurator tool, among others.

Identify Stakeholders Your second activity is to identify the relevant stakeholders (principle PP_1: Identify relevant stakeholders), which 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 implemention. In our exprience, 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 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). In case it is 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 assure that always 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 when 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 familiarizing 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 learn 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 represenation 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 datasets of clone & own-based systems (J. Krueger and Berger, 2020; Strueber et al., 2019; Kuiter et al., 2018).

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 difference 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 help trying out product deriviation (cf. principle QA_2). Engineers can experience whether the derivation feels viable, that is, going through the feature model and making selections to establish a configuration.

The pilot project also helps to, if envisioned, connect business and development worlds. Connecting features to assets and to business aspects is also important, since doing that too late 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. Having selected a reasonably small sub-system for the pilot project can substantually 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 the feature model maintenance and evolution. Since a feature model is brittle, one or 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 277, 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 in customer-facing catalogs, that your stakeholders are familiar with.

(⚙) 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 relevant information about features and their relationships to the subsequent modeling phase. Iterating between both phases allows to gradually increase 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 with obtaining core domain knowledge from the relevant experts. Recall the activity Establish a Forum and a Workshop Format on page 271 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 the 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 strategy, but put more emphasis on either one based on the adoption strategy. You should also recall how a feature is defined (cf. Def. 8.4), 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 usually start 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 arose 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[6], 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 **??** and the relevant literature (Apel, Batory, et al. (2013b), Chapters 4 an 5) for details about implementation techniques for variation points,

[6] https://meldmerge.org

as well for methods and tools to integrate the cloned systems into one (product-line) platform (Assunção et al., 2017; Lillack et al., 2019; Rubin, Krzysztof Czarnecki, and Chechik, 2015; J. Krueger and Berger, 2020). Overall, your task is to "convert" implementation differences to 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 them various times why the difference exists; leading to increasingly domain-oriented explanations, until the difference can be described by the presence or 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 (Kyo Kang et al., 1990) and product-line scoping methods (Schmid, 2000; John and Eisenbarth, 2009; John, Knodel, et al., 2006). According to Krzysztof Czarnecki and U. W. Eisenecker (2000), 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 (Bayer, Flege, et al., 1999), 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 271). 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. These constraints need to reflect the semantics of how you can combine features via the assets they map to in any case. 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 models are 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 (Berger, Nair, et al., 2014b) 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 safe the effort for modeling constraints. Another strategy, seen in practice, is to maintain sets of tested configurations, wich are evolved and maintained together with the model. In contrast, if the main users of a 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. 8.2) and many other systems software projects (Berger, She, Lotufo, Wąsowski, et al., 2013). The complexity of these models and the sheer number of their configurations used for the running systems, demand modeling all constraints.

(✿) Identify Constraints As discussed in the box on page 278, 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 (Nadi et al., 2014; Nadi et al., 2015). Since such analysis techniques are difficult to setup and use, rather the developers should inform the modelers about the constraint, 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 run-time errors*. While details are described by Nadi et al. (2015), 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. (2015). 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 run-time, for instance, when the system crashes due to null-pointer de-referencing or buffer overflows. Notably, they are much more difficult to detect than the 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 rather 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, which 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, called staged configuration (Krzysztof Czarnecki, Helsen, and U. Eisenecker, 2005). Finally, some feature-modeling tools allow to specify soft constraints, such as "recommends" (Berger, Nair, et al., 2014b).

From these sources of constraints, observe that, while the 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 (Lotufo et al., 2010). 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 it 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, the 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 1000 files (cf. Sect. 8.2) is an extreme example. From our experience, all commercial models we have seen with several hundred feaures 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.

Model modularization has two sub-activities:

- **Define Structure of Model Files** To decompose, what you do is to define a hierarchy of feature models, beginning with a root model. This model's top-level features would 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, what you can do is to 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 with creating an initial, coarse hierarchy of features within a feature model (if you created multiple feature models, select the one you think its features are most well-understood).

Start with 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 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. 8.7) 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 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 page 277. 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

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 software 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 also 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, so to avoid meaningless configurations (redundant feature selectins 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 avoid that you can 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 the need for 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 would have many intermediate features, which are usually vague and not very distinct, and as such difficult to understand for stakeholders.

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 into 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 a cost to 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.

Feature Constraints

Constraints restrict the values of features based on other features' values to prevent undesired or invalid system 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 (Nadi et al., 2015):

- Constraints enforce low-level dependencies between software assets, mainly code. Since software systems, especially software 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 run-time 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, shown as menus and sub-menus in a tree-like organization. To foster such an organization, feature model 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 (Apel, Atlee, et al., 2014) arise, which need extra code to handle it. For instance, we observed that in the Linux kernel, when supporting a certain, but 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.

For placing the feature in the hierarchy, consider again the properties given in the box on page 277. 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 feature responsible, or so-called visibility conditions (Berger, She, Lotufo, Wąsowski, et al., 2013), 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 page 273), then conduct this optional activity.

Declaring dependencies between features might require regrouping of features, removing the dependency, or extracting those 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 of a feature to 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 slighty 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 run-time errors are rather seen in cross-tree constraints or feature groups.

(⚙) Define Views In addition to model modularization, some feature-modeling tools allow to create 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 evolution and maintenance, you also want to use the following ways of validation. Notably, the last one, regression testing, is primarily relevant for maintenance and evolution.

- ⚙ **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, as well as they can adivse on feature names and whether their structure in the hierarchy is intuitive. It is also beneficial when experts who did not participate in the modeling take part in the validation—among others, 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. Apparently, 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, also reinforcing 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 (Meinicke et al., 2017), will provide you some automated analysis that tells you whether a change to the model will have an effect on existing configurations. However, these analyses are confined to the feature model, but it is often desired to analyze the effect on the actual variants (Mukelabai et al., 2018). 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, assuring the coverage of feature configurations that cover variants that are in use, ideally at 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 271) come in handy here. Still, while many stakeholders are involved, one or only 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 279).

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 modeled 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 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 279).

8.7 Spectrum of Meta-Modeling

DSLs and feature models are two different techniques that belong to the same continuum of meta-modeling or domain modeling. See the box on page 53 for a discussion on the relation of meta-modeling and domain modeling. This is illustrated in Fig. 8.10. Writing project-specific code (right-most) is the most flexible and the hardest to maintain mechanism for customizing software products. 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). 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, with the cheapest to maintain and the least flexible way to customize a system.

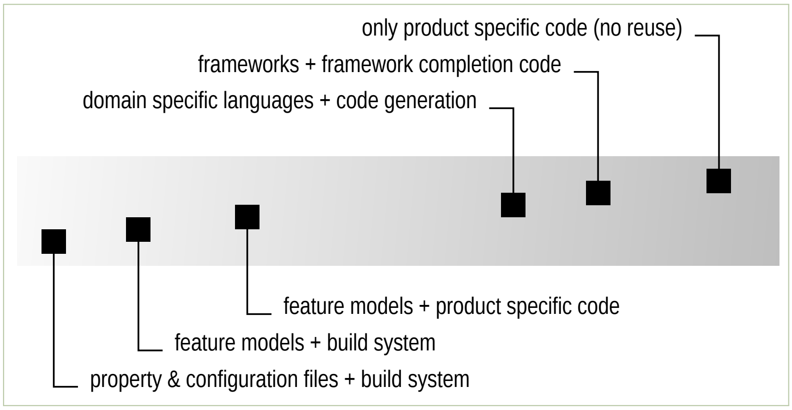

only product specific code (no reuse)
frameworks + framework completion code
domain specific languages + code generation
feature models + product specific code
feature models + build system
property & configuration files + build system

Figure 8.10: *Spectrum of Domain Modeling (Krzysztof Czarnecki and U. Eisenecker, 2000).*

Stahl and Völter (2005) advice 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, explains 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. 8.8, where feature modeling (where you basically switch on or of features) was not sufficient to represent all the possible concrete fire alarm systems, since that domain is rather about instantiating things like fire detectors and alarm devices and connecting them in a topology. To this end, the expressive power of DSLs was necessary.

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.

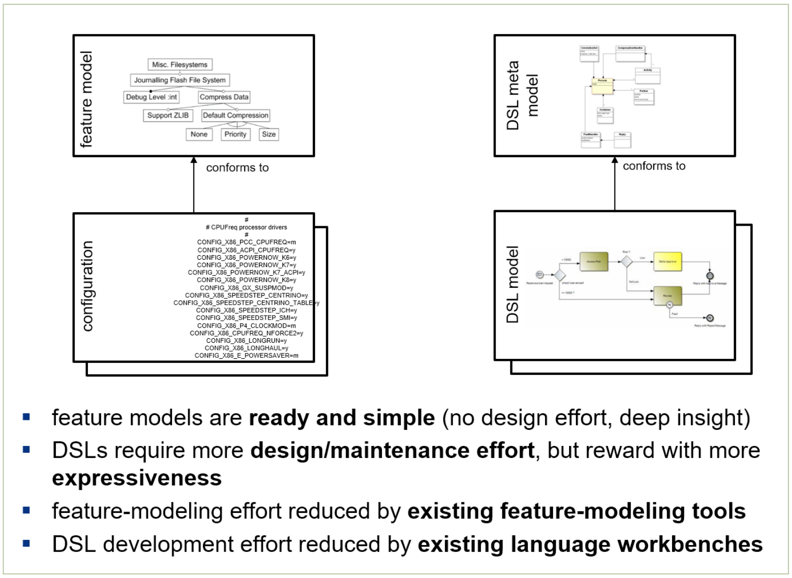
To decrease complexity, the MDSE domain and the MDSE platform should be as close to each other as possible. Ideally, the platform (or framework) should provide implementation of domain concepts.

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 behaviour, 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 existing one, but it increases risks and 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.*

Figure 8.11 illustrates the difference between feature models and DSLs.



```
#
# CPUFreq processor drivers
#
CONFIG_X86_PCC_CPUFREQ=m
CONFIG_X86_ACPI_CPUFREQ=y
CONFIG_X86_POWERNOW_K6=y
CONFIG_X86_POWERNOW_K7=y
CONFIG_X86_POWERNOW_K7_ACPI=y
CONFIG_X86_POWERNOW_K8=y
CONFIG_X86_GX_SUSPMOD=y
CONFIG_X86_SPEEDSTEP_CENTRINO=y
CONFIG_X86_SPEEDSTEP_CENTRINO_TABLE=y
CONFIG_X86_SPEEDSTEP_ICH=y
CONFIG_X86_SPEEDSTEP_SMI=y
CONFIG_X86_P4_CLOCKMOD=m
CONFIG_X86_CPUFREQ_NFORCE2=y
CONFIG_X86_LONGRUN=y
CONFIG_X86_LONGHAUL=y
CONFIG_X86_E_POWERSAVER=m
```

- feature models are **ready and simple** (no design effort, deep insight)
- DSLs require more **design/maintenance effort**, but reward with more **expressiveness**
- feature-modeling effort reduced by **existing feature-modeling tools**
- DSL development effort reduced by **existing language workbenches**

Figure 8.11: *Feature models versus DSLs*

8.8 Case Study: A Fire Alarm System

Let us build a meta-model that allows modeling a fire-alarm installation. It is based on a real project (Berger, Stanciulescu, et al., 2014) we conducted with a Norwegian company producing fire-alarm systems for industry 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 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 8.12 illustrates a simple installation of a fire alarm system.

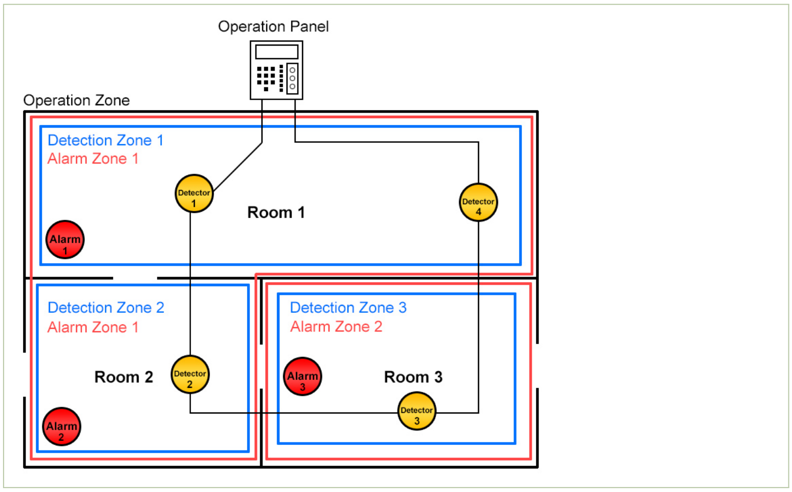


Figure 8.12: *Illustrative example of a simple fire-alarm installation (Berger, Stanciulescu, et al., 2014).*

Motivation Autronica strives for checking 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 in Autronica.

Modeling configurations using a custom modeling tool. Today, Autronica is handling 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 were 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 describing the constraints relating the two.

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 responsibles. 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 one up to 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. Both have 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 having 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 8.13 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.

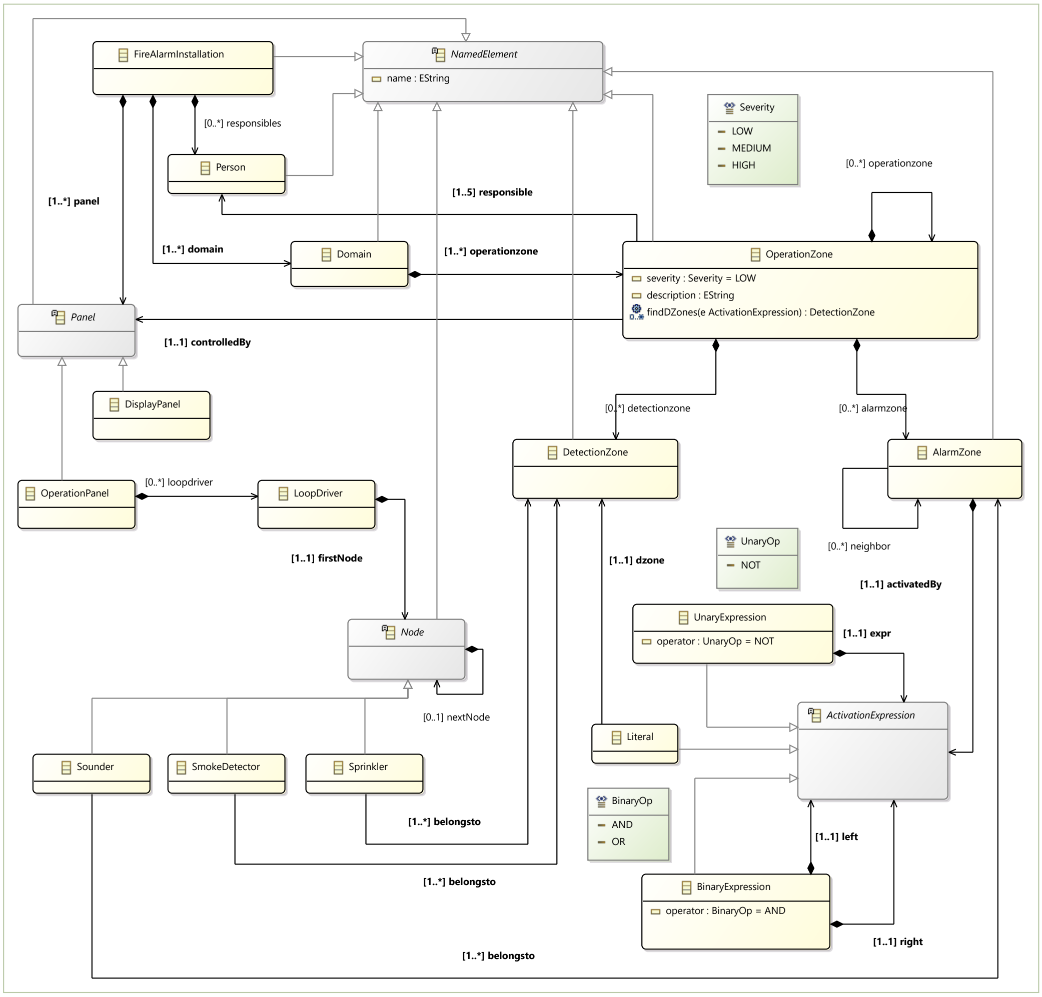


Figure 8.13: *Meta-Model for fire-alarm installations*

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 responsibles (*invariant: twoResponsibles*).

- Each responsible 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 the 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. 8.14, 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. 5.7, such as the tutorial of Cabot and Gogolla (2012). 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 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). Furthermore, note that the *invariant sameDomain* is already enforced by the meta-model and can be omitted.

```
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
```

Figure 8.14: *Additional constraints (static semantics) for our fire-alarm meta-model from Fig. 8.13 defined as OCL constraints*

Further Reading

Classical textbooks on SPLE are those by Apel, Batory, et al. (2013b), P. Clements and L. Northrop (2001), Pohl, Böckle, and F. v. d. Linden (2005), F. J. v. d. Linden, Schmid, and Rommes (2007), and Capilla, Bosch, and K.-C. Kang (2013).

The body of work on feature modeling is humongeous. The FODA report (Kang et al., 1990) is the most popular work on feature oriented domain analysis, which has proposed the feature modeling notation. Another introduction to feature modeling is Chapter 4 of the book by Krzysztof Czarnecki and U. Eisenecker (2000). A brief history of the feature modeling notation is provided by (Berger and Collet, 2019).

Note that the academic feature modeling languages usually come with a graphical syntax, but there are also textual languages tat can be seen as feature-model-like, for instance: TVL (Classen, Boucher, and Heymans, 2011; Hubaux, Boucher, et al., 2011), ClaFeR (Bak, Krzysztof Czarnecki, and Wąsowski, 2010; Bąk et al., 2014), and of course Kconfig (Berger, She, Lotufo, Wąsowski, et al., 2013; She and Berger, 2010) and CDL (Berger, She, Lotufo, Wąsowski, et al., 2013; Berger and She, 2010). A comparison of textual languages is provided by Eichelberger and Schmid (2013)

Clafer (Bak, Krzysztof Czarnecki, and Wąsowski, 2010) is a language that allows seamless switching from feature modeling to structural modeling (class modeling). We are presently working on extending Clafer towards behavior modeling, which would also allow incremental addition of behaviors for mature projects.

Exercises

Exercise 8.2. 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 only by 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. *Hint: Consider a maximum of ten features.*

Exercise 8.3. 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. You remember that 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 8.4. Draw a feature model for the following product line of a (very simple) robot control software.

A robot has always 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 a RGBD-camera. The usage of a RGBD-camera requires the inclusion of a computer. If the parallel gripper is chosen, the biped option is not possible.

Exercise 8.5. 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.

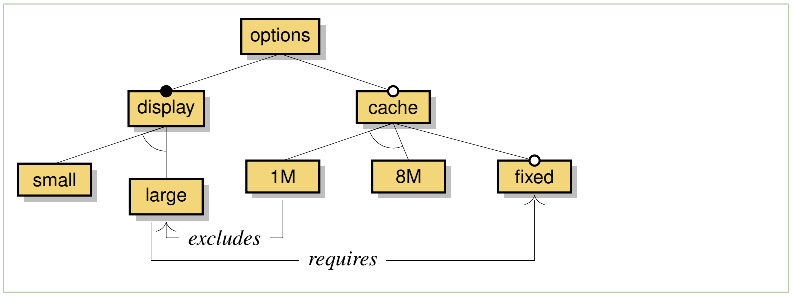


Figure 8.15: A simplified feature model in concrete graphical syntax

Exercise 8.6. Consider the feature model presented in Fig. 8.15. 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 8.7. Consider the following Clafer model.

```
telematicsSystem
  xor channel
    single
    dual

  extraDisplay ?
    xor size
      small
      large
         [ dual  ]
```

Figure 8.16: A simple Clafer model of a car telematics system

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 8.8. Consider the feature model of a car entertainment system presented in Fig. 8.15. 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 to add feature constraints outside the diagram.

Exercise 8.9. Consider the following Clafer model.

```
telematicsSystem
  xor channel
    single
    dual

  extraDisplay ?
    xor size
      small
      large
    [ large => dual ]
```

Figure 8.17: A simple Clafer model of a car telematics system

Change the above model 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). Like in the old model it is still allowed to have either small or large extra display alone, and it is still required to have at least 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)

Exercise 8.10. Discuss the differences between modeling a product line using feature models versus DSLs. List at least two advantages of each.

References

Accioly, Paola, Paulo Borba, and Guilherme Cavalcanti (2018). “Understanding semi-structured merge conflict characteristics in open-source java projects”. In: *Empirical Software Engineering* 23.4, pp. 2051–2085.

Akesson, Jonas et al. (2019). “Migrating the Android Apo-Games into an Annotation-Based Software Product Line”. In: *SPLC*.

Alves, Vander et al. (2010). “Requirements engineering for software product lines: A systematic literature review”. In: *Information and Software Technology* 52.8, pp. 806–820.

Andersson, Jesper and Jan Bosch (2005). “Development and use of dynamic product-line architectures”. In: *IEE Proceedings-Software* 152.1, pp. 15–28.

Apel, Sven, Joanne M. Atlee, et al. (2014). “Feature Interactions: The Next Generation (Dagstuhl Seminar 14281)”. In: *Dagstuhl Reports* 4.7. Ed. by Sven Apel et al., pp. 1–24. ISSN: 2192-5283.

Apel, Sven, Don Batory, et al. (2013a). *Feature- Oriented Software Product Lines*. Springer.

– (2013b). *Feature-Oriented Software Product Lines*. Springer.

Assunção, Wesley K. G. et al. (2017). “Reengineering legacy applications into software product lines: a systematic mapping”. In: *Empirical Software Engineering* 22.6, pp. 2972–3016.

Bak, Kacper, Krzysztof Czarnecki, and Andrzej Wąsowski (2010). “Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled”. In: *SLE*. Ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer, pp. 102–122. ISBN: 978-3-642-19439-9.

Bąk, Kacper et al. (2014). “Clafer: unifying class and feature modeling”. In: *Soft. & Sys. Modeling*, pp. 1–35. ISSN: 1619-1366.

Balzerani, Luca et al. (2005). “A product line architecture for web applications”. In: *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 1689–1693.

Bayer, Joachim, Oliver Flege, et al. (1999). “PuLSE: A methodology to develop software product lines”. In: *Proceedings of the 1999 symposium on Software reusability*, pp. 122–131.

Bayer, Joachim, Thomas Forster, et al. (2006). “Process Family Engineering in Automotive Control Systems: a Case Study”. In: *GPCE*.

Berger, Thorsten and Philippe Collet (2019). “Usage Scenarios for a Common Feature Modeling Language”. In: *First International Workshop on Languages for Modelling Variability (MODEVAR)*.

Berger, Thorsten, Daniela Lettner, et al. (2015). “What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines”. In: *SPLC*.

Berger, Thorsten, Divya Nair, et al. (2014a). “Three Cases of Feature-Based Variability Modeling in Industry”. In: *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

– (2014b). “Three Cases of Feature-Based Variability Modeling in Industry”. In: *MODELS*.

Berger, Thorsten, Rolf-Helge Pfeiffer, et al. (2014). “Variability Mechanisms in Software Ecosystems”. In: *Information and Software Technology* 56.11, pp. 1520–1535.

Berger, Thorsten, Ralf Rublack, et al. (2013). “A survey of variability modeling in industrial practice”. In: *VaMoS*.

Berger, Thorsten and Steven She (2010). *Formal Semantics of the CDL Language*. Tech. rep. Technical Note. Department of Computer Science, University of Leipzig, Germany. URL: http://www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

Berger, Thorsten, Steven She, Rafael Lotufo, Krzysztof Czarnecki, et al. (2010). "Feature-to-Code Mapping in Two Large Product Lines". In: *SPLC*.

Berger, Thorsten, Steven She, Rafael Lotufo, Andrzej Wąsowski, et al. (2013). "A Study of Variability Models and Languages in the Systems Software Domain". In: *IEEE Transactions on Software Engineering* 39.12, pp. 1611–1640.

Berger, Thorsten, Stefan Stanciulescu, et al. (2014). "To Connect or Not to Connect: Experiences from Modeling Topological Variability". In: *SPLC*.

Berger, Thorsten, Jan-Philipp Steghöfer, et al. (2019). "The State of Adoption and the Challenges of Systematic Variability Management in Industry". In: *Empirical Software Engineering*. Preprint.

Beuche, Danilo (2004). "pure::variants Eclipse Plugin". User Guide. pure-systems GmbH. Available from http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf.

Bosch, Jan (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education.

– (2009). "From software product lines to software ecosystems". In: *Proceedings of the 13th International Software Product Line Conference*. SPLC '09.

Businge, John et al. (2018). "Clone-Based Variability Management in the Android Ecosystem". In: *ICSME*.

Cabot, Jordi and Martin Gogolla (2012). "Object Constraint Language (OCL): A Definitive Guide". In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Vol. 7320. Lecture Notes in Computer Science. Springer, pp. 58–90. DOI: 10.1007/978-3-642-30982-3_3. URL: https://doi.org/10.1007/978-3-642-30982-3_3.

Capilla, Rafael, Jan Bosch, and Kyo-Chul Kang (2013). "Systems and Software Variability Management: Concepts, Tools and Experiences". In:

Chastek, Gary et al. (2011). "Engineering a Production Method for a Software Product Line". In: *Proceedings of the 2011 15th International Software Product Line Conference*. SPLC '11. Washington, DC, USA: IEEE Computer Society, pp. 277–286. ISBN: 978-0-7695-4487-8. DOI: 10.1109/SPLC.2011.46.

Chen, L. and M. Ali Babar (2009). "A Survey of Scalability Aspects of Variability Modeling Approaches". In: *Workshop on Scalable Modeling Techniques for Software Product Lines at SPLC*.

Chen, Lianping, Muhammad Ali Babar, and Nour Ali (2009). "Variability management in software product lines: a systematic review". In: *SPLC'09*.

Classen, Andreas, Quentin Boucher, and Patrick Heymans (2011). "A text-based approach to feature modelling: Syntax and semantics of TVL". In: *Science of Computer Programming* 76.12, pp. 1130–1143. ISSN: 0167-6423. DOI: 10.1016/j.scico.2010.10.005.

Classen, Andreas, Patrick Heymans, and Pierre-Yves Schobbens (2008). "What's in a Feature: A Requirements Engineering Perspective". In: *FASE*.

Clements, P. and L. Northrop (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Clements, Paul and John Bergey (2005). *The US Army's Common Avionics Architecture System (CAAS) Product Line: A Case Study*. Tech. rep. Software Engineering Institute, Carnegie Mellon University.

Czarnecki, K. et al. (2012). “Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches”. In: *VaMoS*.

Czarnecki, Krzysztof and Ulrich Eisenecker (2000). *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley.

Czarnecki, Krzysztof and Ulrich W. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley.

Czarnecki, Krzysztof, Simon Helsen, and Ulrich Eisenecker (2005). “Staged configuration through specialization and multilevel configuration of feature models”. In: *Software process: improvement and practice* 10.2, pp. 143–169.

Debbiche, Jamel et al. (2019). “Migrating the Java-Based Apo-Games into a Composition-Based Software Product Line”. In: *SPLC*.

Dhungana, Deepak, Patrick Heymans, and Rick Rabiser (2010). “A Formal Semantics for Decision-oriented Variability Modeling with DOPLER”. In: *VaMoS*.

Dordowsky, Frank and Walter Hipp (2009). “Adopting Software Product Line Principles to Manage Software Variants in a Complex Avionics System”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09. San Francisco, California, USA: Carnegie Mellon University, pp. 265–274. URL: http://dl.acm.org/citation.cfm?id=1753235.1753272.

Dubinsky, Yael et al. (2013). “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *CSMR*.

Duc, Anh Nguyen et al. (2014). “Forking and Coordination in Multi-platform Development: A Case Study”. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’14.

Dziobek, Christian et al. (2008). “Functional Variants Handling in Simulink Models”. In: *MACDE*. URL: https://www.researchgate.net/publication/238778580_Functional_Variants_Handling_in_Simulink_Models.

Eichelberger, Holger and Klaus Schmid (2013). “A systematic analysis of textual variability modeling languages”. In: *Proceedings of the 17th International Software Product Line Conference*.

Eklund, Ulrik and Håkan Gustavsson (2013). “Architecting automotive product lines: Industrial practice”. In: *Science of Computer Programming* 78.12, pp. 2347–2359.

El-Sharkawy, Sascha, Adam Krafczyk, and Klaus Schmid (2015). “Analysing the Kconfig semantics and its analysis tools”. In: *GPCE*.

Flores, Rick, Charles Krueger, and Paul Clements (2012). “Mega-Scale Product Line Engineering at General Motors”. In: *Proc. SPLC*.

Fogdal, Thomas et al. (2016). “Ten years of product line engineering at Danfoss: lessons learned and way ahead”. In: *SPLC*.

Ganesan, Dharmalingam et al. (2009). “Verifying Architectural Design Rules of the Flight Software Product Line”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09. San Francisco, California, USA: Carnegie Mellon University, pp. 161–170.

Ganz, Christopher and Michael Layes (1998). “Modular turbine control software: A control software architecture for the ABB gas turbine family”. In: *International Workshop on Architectural Reasoning for Embedded Systems*.

Garcia, Sergio, Daniel Strueber, Davide Brugali, Thorsten Berger, et al. (2020). “Robotics Software Engineering: A Perspective from the Service Robotics Domain”. In: *28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*.

Garcia, Sergio, Daniel Strueber, Davide Brugali, Alessandro Di Fava, et al. (2019). "Variability Modeling of Service Robots: Experiences and Challenges". In: *13th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*.

Gousios, Georgios, Martin Pinzger, and Arie van Deursen (2014). "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, pp. 345–355.

Gustavsson, Hakan and Ulrik Eklund (2010). "Architecting Automotive Product Lines: Industrial Practice". In: *SPLC*.

Habli, Ibrahim and Tim Kelly (2007). "Challenges of Establishing a Software Product Line for an Aerospace Engine Monitoring System". In: *Proceedings of the 11th International Software Product Line Conference*. SPLC '07. Washington, DC, USA: IEEE Computer Society, pp. 193–202. ISBN: 0-7695-2888-0. DOI: 10.1109/SPLC.2007.14.

Hardung, Bernd, Thorsten Kölzow, and Andreas Krüger (2004). "Reuse of Software in Distributed Embedded Automotive Systems". In: *Proceedings of the 4th ACM International Conference on Embedded Software*. EMSOFT '04.

Hess, Klaus-Dieter and Frank Dordowsky (2008). "Rational ClearCase migration to a complex avionics project - an experience report". In: *CONQUEST*.

Hubaux, Arnaud, Quentin Boucher, et al. (2011). "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, pp. 337–356. ISBN: 978-3-642-19439-9.

Hubaux, Arnaud, Yingfei Xiong, and Krzysztof Czarnecki (2012). "A User Survey of Configuration Challenges in Linux and ECos". In: *VaMoS*.

Janota, Mikolás, Victoria Kuzina, and Andrzej Wąsowski (2008). "Model Construction with External Constraints: An Interactive Journey from Semantics to Syntax". In: *MoDELS*. Ed. by Krzysztof Czarnecki et al. Vol. 5301. Lecture Notes in Computer Science. Springer, pp. 431–445. ISBN: 978-3-540-87874-2.

Jansen, Slinger, Anthony Finkelstein, and Sjaak Brinkkemper (2009). "A sense of community: A research agenda for software ecosystems". In: *31st International Conference on Software Engineering - Companion Volume*. IEEE, pp. 187–190.

Ji, Wenbin et al. (2015). "Maintaining Feature Traceability with Embedded Annotations". In: *SPLC*.

John, Isabel and Michael Eisenbarth (2009). "A decade of scoping: a survey". In: *Proceedings of the 13th International Software Product Line Conference*, pp. 31–40.

John, Isabel, Jens Knodel, et al. (2006). "A practical guide to product line scoping". In: *10th International Software Product Line Conference (SPLC'06)*. IEEE, pp. 3–12.

Kang et al. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21.

Kang, K.C. (2009). "FODA: Twenty Years of Perspective on Feature Models". In: *Keynote Address at the 13th International Software Product Line Conference*. SPLC.

Kang, Kyo et al. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. SEI, CMU.

Kleene, S. C. (1938). “On Notation for Ordinal Numbers”. In: *The Journal of Symbolic Logic* 3.4, pp. 150–155. ISSN: 00224812.

Koziolek, Heiko et al. (2016). “Assessing software product line potential: an exploratory industrial case study”. In: *Empirical Software Engineering* 21.2, pp. 411–448.

Krueger, Charles (2002). “Variation Management for Software Production Lines”. In: *Proceedings of the Second International Conference on Software Product Lines*. SPLC 2.

Krueger, Charles W. (2007). “BigLever software gears and the 3-tiered SPL methodology”. In: *OOPSLA*.

Krueger, Jacob and Thorsten Berger (2020). “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)*.

Krueger, Jacob, Wanzi Gu, et al. (2018). “Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin”. In: *VaMoS*.

Krüger, Jacob et al. (2018). “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.

Krüger, Jacob et al. (2019). “Where is my feature and what is it about? a case study on recovering feature facets”. In: *Journal of Systems and Software* 152, pp. 239–253.

Kuiter, Elias et al. (2018). “Getting rid of clone-and-own: moving to a software product line for temperature monitoring”. In: *SPLC*.

Liang, Liang, Zhiqiang Hu, and Xiangyun Wang (2005). “An open architecture for medical image workstation”. In: *Medical Imaging 2005: PACS and Imaging Informatics*.

Lillack, Max et al. (2019). “Intention-Based Integration of Software Variants”. In: *41st International Conference on Software Engineering*. ICSE.

Linden, Frank J. van der, Klaus Schmid, and Eelco Rommes (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

Linden, Frank van der (2002). “Software Product Families in Europe: The Esaps & Café Projects”. In: *IEEE Software* 19.4, pp. 41–49.

Lotufo, Rafael et al. (2010). “Evolution of the Linux Kernel Variability Model”. In: *SPLC*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, pp. 136–150. ISBN: 978-3-642-15578-9.

Mahmood, Wardah et al. (2020). “Causes of Merge Conflicts: A Case Study of ElasticSearch”. In: *14th International Working Conference on Variability Modelling of Software-intensive Systems (VaMoS)*.

McKee, S. et al. (2017). “Software Practitioner Perspectives on Merge Conflicts and Resolutions”. In: *ICSME*.

Mecklenburg, Robert (2004). *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. " O’Reilly Media, Inc.".

Meinicke, Jens et al. (2017). *Mastering Software Variability with FeatureIDE*. Springer.

Menezes, Gleiph Ghiotto Lima de (Dec. 2016). “On the nature of software merge conflicts”. PhD thesis. Federal Fluminense University.

Mohagheghi, Parastoo and Reidar Conradi (June 2008). “An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product”. In: *ACM Trans. Softw. Eng. Methodol.* 17.3, 13:1–13:31. ISSN: 1049-331X. DOI: 10.1145/1363102.1363104. URL: http://doi.acm.org/10.1145/1363102.1363104.

Mojica, Israel J. et al. (Mar. 2014). “A Large Scale Empirical Study on Software Reuse in Mobile Apps”. In: *IEEE Software* 31.2, pp. 78–86.

Mukelabai, Mukelabai et al. (2018). “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)*.

Nadi, Sarah et al. (2014). “Mining Configuration Constraints: Static Analyses and Empirical Results”. In: *ICSE*.

– (2015). “Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study”. In: *IEEE Transactions on Software Engineering*. preprint.

Nesic, Damir et al. (2019). “Principles of Feature Modeling”. In: *FSE*.

Northrop, Linda M. (2010). “Introduction to Software Product Lines”. In: *SPLC*.

Obbink, H. et al. (2000). “COPA: a component-oriented platform architecting method for families of software-intensive electronic products”. In: *Tutorial for SPLC*.

Object Management Group (2017). *Unified Modeling Language Specification 2.5.1*. https://www.omg.org/spec/UML.

Parnas, David (July 1976). “On the design and development of program families”. In: *IEEE Transactions on Software Engineering* SE-2.1, pp. 1–9.

Passos, Leonardo et al. (2018). “A Study of Feature Scattering in the Linux Kernel”. In: *IEEE Transactions on Software Engineering*. Preprint.

Pohjalainen, Pietu (2011). “Bottom-up Modeling for a Software Product Line: An Experience Report on Agile Modeling of Governmental Mobile Networks”. In: *Proceedings of the 2011 15th International Software Product Line Conference*. SPLC’11.

Pohl, Klaus, Günter Böckle, and Frank J. van der Linden (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

Pohl, Klaus, Günter Böckle, and Frank van der Linden (2005). *Software Product Line Engineering*. Springer Verlag.

Riebisch, Matthias (2003). “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.

Rösel, Andreas (1998). “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*.

Rubin, Julia, Krzysztof Czarnecki, and Marsha Chechik (2015). “Cloned product variants: from ad-hoc to managed software product lines”. In: *STTT* 17.5, pp. 627–646.

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Pearson Education Limited.

Schmid, Klaus (2000). “Scoping software product lines”. In: *Software Product Lines*. Springer, pp. 513–532.

Schmid, Klaus, Rick Rabiser, and Paul Grünbacher (2011a). “A Comparison of Decision Modeling Approaches in Product Lines”. In: *VAMoS*.

– (2011b). “A comparison of decision modeling approaches in product lines”. In: *VaMoS*, pp. 119–126.

Schobbens, P.-Y. et al. (2006). “Feature Diagrams: A Survey and a Formal Semantics”. In: *Proc. RE*.

Sharp, David C (1998). “Reducing avionics software cost through component based product line development”. In: *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No. 98CH36267)*.

She, Steven and Thorsten Berger (2010). *Formal Semantics of the Kconfig Language*. Technical Note. Available at https://gsd.uwaterloo.ca/sites/default/files/kconfig_semantics.pdf.

Sinnema, Marco and Sybren Deelstra (2007). “Classifying variability modeling techniques”. In: *Information and software technology* 49.7, pp. 717–739.

Software Engineering Institute (n.d.). *Catalog of Software Product Lines*. http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm.

Reuse-Driven Software Processes Guidebook, Version 02.00.03 (1993). Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC.

Stahl, Thomas and Markus Völter (2005). *Model-Driven Software Development*. Wiley.

Stanciulescu, Stefan, Sandro Schulze, and Andrzej Wąsowski (2015). “Forked and Integrated Variants in an Open-Source Firmware Project”. In: *ICSME*.

Staples, Mark and Derrick Hill (2004). “Experiences Adopting Software Product Line Development without a Product Line Architecture”. In: *APSEC*.

Stoll, Pia et al. (2009). “Supporting Usability in Product Line Architectures”. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC ’09.

Strueber, Daniel et al. (2019). “Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems”. In: *23rd International Systems and Software Product Line Conference (SPLC)*.

Svahnberg, Mikael and Jan Bosch (Nov. 1999). “Evolution in Software Product Lines: Two Cases”. In: *Journal of Software Maintenance* 11.6, pp. 391–422. ISSN: 1040-550X.

Takebe, Yasuaki et al. (2009). “Experiences with Software Product Line Engineering in Product Development Oriented Organization”. In: *SPLC*.

Thüm, Thomas et al. (2014). “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79, pp. 70–85.

Tischer, Christian et al. (2011). “Experiences from a Large Scale Software Product Line Merger in the Automotive Domain”. In: *SPLC*.

Verlage, Martin and Thomas Kiesgen (2005). “Five years of product line engineering in a small company”. In: *ICSE*.

Ziadi, Tewfik, Loïc Hélouët, and Jean-Marc Jézéquel (2004). “Towards a UML Profile for Software Product Lines”. In: *Software Product-Family Engineering*.

Zippel, Roman (2017). *KConfig*. Technical Documentation. Available at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

A Class Modeling

We use class modeling as the main meta-modeling formalism in the book—with an occasional use of functional data types and feature models, which are other popular alternatives. In this chapter, we recall the main aspects of class modeling.

UML (Unified Modeling Language) provides a complete set of notations for modeling software systems. These can be used for modeling many different aspects, including requirements, architecture, types, structure, and behaviors.

In this book, we primarily use UML *class diagrams*, and only for structural modeling of the abstract syntax of a language. As such, our view of UML is clearly restricted. We also mix-in some non-standard extensions included in the Eclipse Modeling Framework EMF with its class-modeling language Ecore.

A.1 Classes and Objects

Let us start with a definition of a class:

Definition A.1. *A* class *is an abstraction that specifies attributes of a set of concept instances (objects) and their relations to other sets of concept instances (objects).*

Importantly, in this book, a class is *not* a programming language concept that can be identified in the system source code. It is not a program type (or this is not its primary aspect). We first and foremost will be using classes to model real-world concepts or—more precisely—*domain-level* concepts. These are often much more abstract than what Java or C# classes are. Our classes will rarely map one-to-one to implementation classes. This big leap from implementation to a conceptual application domain is necessary in order to obtain the productivity gains promised by domain-specific languages and MDSE.

Both *classes* and *objects* are depicted by boxes with three compartments: names, attributes, and operations. When visualizing models, attributes and operations can be omitted if they are not essential. In high-level modeling, we usually ignore the class operations. Operations are useful when describing low-level implementation aspects (such as APIs), but they are rarely used in DSL design—thus, you will not see them in this book. At the same time, names and attributes are essential for us.

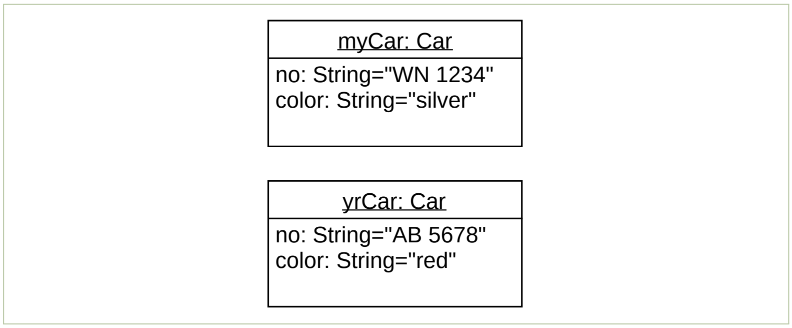


Figure A.1: An object diagram with two objects

info:generalization

Inheritance vs Generalization

You have probably realized that we prefer to use the term *generalization* over the, possibly more common, *inheritance*. This is not without a reason. The modeling experts prefer the former term, because it captures more precisely the meaning of the *kind-of* relationship in concept modeling: it states that one concept is more general than the other, and the latter is a specialization of the former. *Inheritance* is a particular implementation mechanism for generalization when this relationship has to be represented at runtime in an interpreter or generated code (we implement generalization by inheriting attributes and operations). This is the reason why programming language experts, who are typically also compiler writers, would tend to prefer the term *inheritance*. You can safely assume that the two terms are synonyms.

Consider the *object diagram* in Fig. A.1 (object diagrams are called instance specification diagrams in newer versions of UML). It states that there exist two car objects, each with two attributes. In order to distinguish object diagrams from class diagrams, the object names (instance names) are underlined and followed by a colon with the class name the object is an instance of.

In the simplest view, classes are just types for objects. Our objects in Fig. A.1 are of type Car:

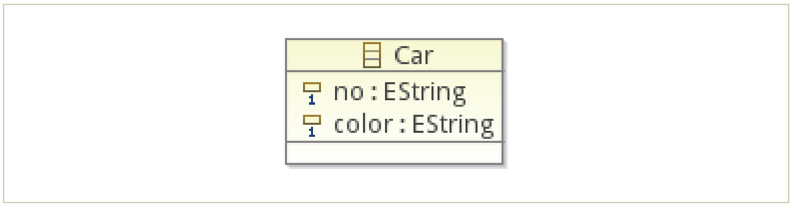


Figure A.2: An example class

The block above represents the class Car. The name is not underlined, and attributes have types, not values. They also have a *multiplicity* constraint (here simply “1”, which says that both attributes are mandatory for any instance of this class). By convention, class names are capitalized, and instance names are not.

A.2 Generalization

A *generalization relation* (also known as *inheritance*) specifies that instance sets of two classes are included:

Definition A.2. *A class A* generalizes *class B, if each instance of B is also an instance of A.*

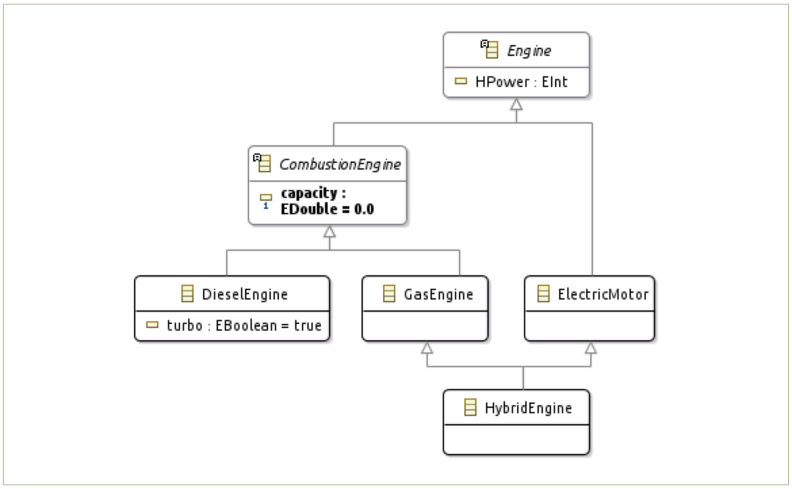


Figure A.3: An example generalization hierarchy of car engine designs

We illustrate the generalization hierarchy using a model of the design space of car engines. In the diagram (Fig. A.3) we read that, among others, every *CombustionEngine* is an *Engine*, and so is every ElectricEngine. A HybridEngine is both a gas and electric engine. We sometimes say that a generalization relation expresses the *kind-of* or an *is-a* relationship: "an electric motor *is a kind of* engine."

The diagram in Fig. A.3 contains two examples of abstract classes (for instance *CombustionEngine*). An abstract class has no instances of its own. In the diagrams, we mark abstract classes using a cursive font for the class name.

Finally, Fig. A.3 also includes a case of multiple-inheritance: a hybrid engine is both a gas engine and an electric motor.[1] Multiple inheritance often appears in meta-modeling applications, when a concept shares features properties with more than one concepts, or when a concept can appear in more than one role.

Class diagrams also allow the modeling of interfaces—this is done by adding an interface property to a class. Note that Eclipse EMF's class-diagram editor puts a little interface icon next to the box label, as shown in Fig. A.4.

[1] Technically, we could argue that this modeling is incorrect. In reality, a hybrid engine is not a specialization of a gas engine and an electric motor, but a *composition* of both—see below for the composition relation.

InterfaceName

Figure A.4: EMF's syntax for interfaces

When EMF generates code, interfaces are mapped to Java interfaces, while classes are mapped both to classes and interfaces. The latter is a simple pattern (a workaround, if you prefer) for Java's lack of multiple inheritance.

A.3 Simple Types

EMF provides simple types (for example the **EString** used above), which are mapped to Java types during code generation. An attribute declaration can be followed by a default value, as in: `color : EString = "red"`.

Enumerations are used to capture a small finite number of discrete simple values of an attribute, for instance a color, as shown in Fig. A.5.

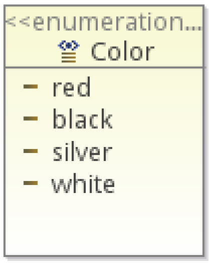


Figure A.5: EMF syntax for enumerations

Enumerations can be used as types for attributes, but cannot be the ends of associations (explained shortly), which is reserved for classes. Fig. A.6 shows the usage of the enumeration Color as the type for the attribute color in the abstract class Vehicle.

It is also possible to introduce new basic types by providing their Java implementations. Details about this mechanism can be found in Budinsky et al., 2004 (search for **EDataType** in the index).

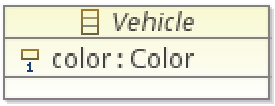


Figure A.6: Using an enumeration as an attribute type

A.4 Associations

An *association* represents a relation between instances of two classes. Note that association is qualitatively different from generalization. We use associations to model all other kinds of relations between objects than *kind-of*. Associations can be bidirectional, then they have no arrows on the ends, or uni-directional (indicated by an arrow). An example of the latter is shown in Fig. A.7.

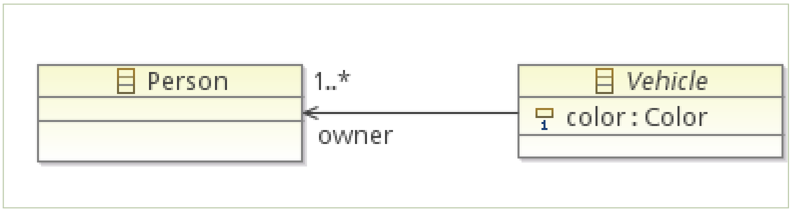


Figure A.7: *A single directional association (also known as a reference)*

The navigable name of the association (the name used for navigation in transformation and constraint code) is written on the "far end." For example, the `myCar.owner` gives the object representing the owner of `myCar`. Note that the owner label is on the other side of the vehicle class—the intuition is that it shows the name that we can use in the context of *Vehicle* to name the associated person object.

In the example, the reference is also decorated with a multiplicity constraint `1..*`, meaning that a vehicle must have at least one owner. More than one owner is allowed in the example (for modeling co-ownership). This also means that technically, `myCar.owner` returns a collection and not a single instance.

In EMF, associations are unidirectional binary references. Unidirectional references can only be navigated in one direction. In UML, references can be bidirectional and *n*-ary. For our purpose of meta-modeling, binary references are typically sufficient. Higher-arity references can be always handled by creating an explicit class that will reify the association (similar to UML association classes). But, as already said, we rarely need it in language design.

Bidirectional references can be simulated using two unidirectional references. EMF allows to link two unidirectional references using the EOpposite property of the reference. In such a case, the generated code maintains links in both directions: whenever you add a link in one direction, the link in the other direction is updated automatically. The mechanism is a bit complicated, and has shortcomings, so test well when you rely on it. In particular, a reference cannot be EOpposite to itself,[2] and special care might be needed if you use references of multiplicity higher than 1.

[2]This sounds a bit complicated, but in fact it appears in real domains for symmetric associations between objects of the same class. For example, consider a class Person and a unidirectional reference marriedTo. One way to model this in EMF would be to make the reference a bidirectional association, but this would require that it becomes an EOpposite of itself, which is not supported.

A.5 Containment (Part-Of)

Associations can be used to denote a *part-of* relation (in contrast to the *kind-of* relation of generalization). This is denoted using a black diamond on the owner side, as shown in Fig. A.8. In this example, we state that each *Vehicle* contains four *Wheel*s as its integral part. This means that a *Vehicle* instance without four wheels cannot exist (such an instance is not well-formed). When a *Vehicle* object is deallocated, the objects representing wheels are also removed.

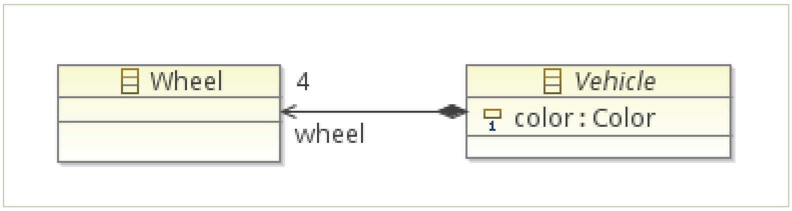


Figure A.8: *A containment (part-of) relation*

The black-diamond symbol is the UML syntax (notation) for the part-of associations. The black-diamond semantics is that every object in such a relation can only have one owner—so there could not be cars sharing wheels, and that the owned objects exist only with the owner. Finally, objects cannot be owners of themselves, so a directed sub-graph of an object diagram (instance), in which all the links instantiate containment edges must be a tree (or a forest).

The black-diamond associations are interchangeably called “compositions,” “aggregations,” and ”part-of relations.” Out of these names, we find “part-of” and “containment” most intuitive, and this is why we use them in the main text of this book.

A.6 Views on Class Models

So far it was not evident that we distinguish *class models* from *class diagrams*. The difference is important, though, when discussing modeling in detail. Diagrams are mere *views on models*, thus, a diagram might only be showing a fragment of a model, and multiple diagrams can show overlapping fragments. The model is usually identified as the collection of all model elements (all classes and relationships). Tools, including Eclipse EMF, typically show models as abstract syntax trees and allow constructing diagrams for parts of these trees.

Two views on class models are particularly interesting: a *taxonomy* and *partonomy*. The taxonomy view is consistent with the standard use of this term in knowledge classification:

Definition A.3. *A* taxonomy *of a class model is a diagram presenting specialization-generalization relations (kind-of relations) between the classes of this model.*

Fig. A.3 presents an example of a taxonomy. A generalization view is always a directed acyclic graph containing only classes and generalization arrows. This graph may be disconnected if we have unrelated concepts in the model's taxonomy.

In general knowledge classification a partonomy is *a hierarchy that deals with part–whole relationships* (after Wiktionary). The use of this term in class modeling is consistent with the general definition, as well, by relating parts using the composition associations:

Definition A.4. *A* partonomy *of a class model is a diagram (a view) presenting only the part-of relationships between classes, so a view presenting the composition associations and classes.*

A partonomy view is always a tree (or more generally a forest if we have classes that are not associated using composition)—this is due to the semantics of containment which disallows sharing of subtrees of the partonomy (no class can be a part of to disjoint containers). Most modeling tools, including Eclipse EMF, requires a single, connected partonomy hierarchy, thus forest partonomy are rarely seen in practice of class modeling. A single partonomy is usually achieved by creating a root class containing (via composition associations) all roots of otherwise disconnected partonomies.

Class diagrams share a lot of commonality with entity-relationship modeling (E/R) used to specify database schema. One key difference introduced by class diagrams was including the taxonomy and partonomy in the model. None of these were part of the original E/R model.

Further Reading

If you have never been exposed to class diagrams, we recommend the book of Fowler (2004), but many other books on UML would be fine, too.

An excellent resource that does not only explain UML class diagrams, but many other types of UML diagrams as well, is https://www.uml-diagrams.org. Its page on class diagrams[3] explains class diagrams and instance specifications. Remember that the latter are not modeled separately in an object diagram anymore, which is deprecated, but directly in the class diagram. We discuss this in Sect. 3.9 in the main part of the book.

Furthermore, many reference guides (a.k.a., *cheat sheets*) exist that provide a brief overview on class diagrams, such as DZone's Refcardz at https://dzone.com/refcardz/getting-started-uml. Jordi Cabot provides an overview on such cheat sheets at https://modeling-languages.com/best-uml-cheatsheets-and-reference-guides.

Finally, since we use Ecore as the class-modeling language of our choice, we also recommend directly looking at Appendix B and the list of further reading in that appendix.

[3] https://www.uml-diagrams.org/class-diagrams-overview.html

Exercises

Exercise A.1. Change the model of Fig. A.3 to more properly reflect the fact that a hybrid engine is not a refinement of a combustion engine and an electric motor, but has both of these as parts combined.

Exercise A.2. *A family consists of persons. Each person may be married to another person. Each person may have a parent, and each parent may have multiple children. Each person has exactly one name, exactly one age and exactly one person number (a unique ID of type String). Each person may be enrolled in a university. University must own one or more study programs.*

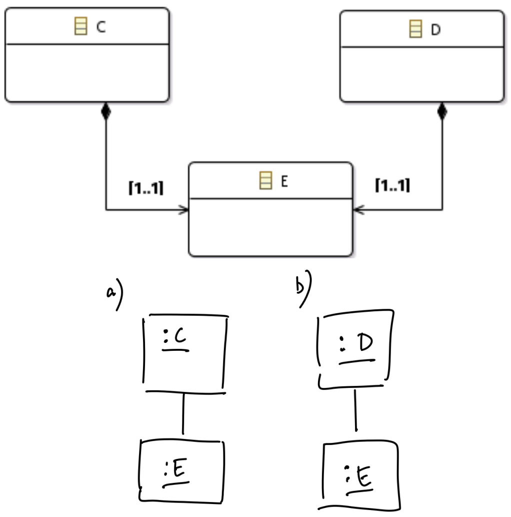
a) Create a simple class model using the tree editor of Eclipse following this description of a domain:

b) Create a valid instance of your diagram representing Bob married to Alice, with their son Sam enrolled in the "SDT programme" of "IT University".

c) For pedagogical reasons we recommend using the tree editor and understanding the relation between the tree editor and the diagram editor in Eclipse (or any other modeling tool you are using). This will yield useful intuitions when we start to build abstract syntax trees of models in the book.

d) Explore different views: Create three diagrams for your model, a complete diagram (that contains all model elements), a diagram only showing the family relations without the enrollment aspects, and a diagram showing university enrollment without family aspects.

Exercise A.3. Consider the following example class diagram.



This diagram is valid in the sense that we can construct its instances. Two example instances are shown as simple instance specification diagrams to the right (a–b). Now consider the three unrelated class diagrams that follow. For each of the diagrams decide whether it is valid (well-formed). If it is valid, draw an example of a non-empty instance diagram. If invalid, explain why.

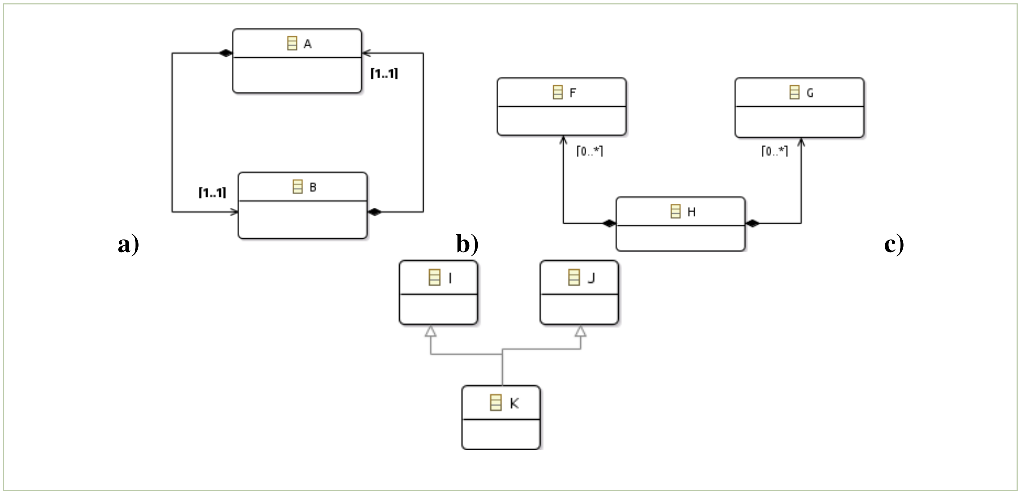


Figure A.9

Exercise A.4. In this exercise (a mini-project, in fact) we use class modeling as a method for system comprehension. Recall (Sect. 1.3) that using models in software development fosters knowledge conservation and reuse by improving the domain understanding as a key strength.

We will use the implementation of JUnit 4 framework as a case study. We assume that you are familiar with unit testing using JUnit, which will make the exercises easier.

To avoid excessive use of time, for this and the following three exercises (that should be solved in order) we bound the time to be used on them. This should give you an impression of the expected level of details. Read exercises A.4–A.6 entirely, before starting to solve this one.

Build a conceptual model of JUnit as a class model, based on available user oriented documentation. Start with reading the user oriented documentation of JUnit: http://junit.sourceforge.net/doc/cookbook/cookbook.htm or https://github.com/junit-team/junit/wiki, but do ignore the javadoc for the time being. The high-level documentation will give names of the key concepts, which will likely translate to a handful of class names and relations between them. These concepts do not necessarily correspond to low level implementation classes precisely.

Identify key concepts, objects, subsystems and record them as classes, associations, generalizations, and aggregations. For example when you find the concept of Test, create the corresponding class. Then you encounter a concept of a Suite that aggregates multiple tests. You can create a Suite class, and make it own one or more tests using composition.

Record cardinalities precisely in your model. If you, at any point, encounter constraints, dependencies between concepts, which cannot be expressed using class diagrams, then note them down in English, either in a separate file, or in an annotation. They will be input for exercises on constraints.

All modeling should be done using a modeling tool (not on paper, not using a drawing tool). An indicative model size is circa 12 classes. Estimated time: approximately 1 hour.

Obviously, this small project can be run on other frameworks than JUnit, be it other implementations of unit testing frameworks, or any other software projects.

Exercise A.5. In continuation of the above exercise, perform a cursory analysis of the developer[a] oriented documentation of JUnit to refine your model. Developer documentation for Junit is essentially only javadoc, available at: http://junit.org/javadoc/latest/. Start with concepts that seem to be already connected to elements in your model. When you study them, refine the model appropriately. Do not mean converting your model to an implementation level model, just codifying classes in JUnit's source code. Rather try adding further abstract concepts and relations to your existing high-level model.

Do not grow your model too much. Focus on understanding whether the selection of classes, associations and generalizations is correct (so whether the lower level documentation confirms your initial sketch from the previous point). Also try to understand and record any constraints (including cardinalities) that you might have spotted.

The objective is not to create a diagram of implementation classes. So there does not have to be (and should not be!) a one-to-one mapping between your model and the classes in the JUnit implementation. We only look at lower level artifacts to understand details of the system that were too hard to understand from user documentation. Estimated time: 1–2 hours.

Exercise A.6. In this exercise, we delve into the JUnit code. Hopefully, after the first two steps, reading the JUnit code is relatively easier. JUnit is a small and well implemented framework, done by some of the best programmers out there. Whenever you get frustrated, remember that it is by orders of magnitude better experience to read it than reading any code you might need to inspect or understand in your job.

Obtain the source code from JUnit GitHub repository (clone https://github.com/junit-team/junit.git) For pedagogical reasons, it is better to work with a stable release, than with a snapshot code that may be buggy. Switch to a stable released branch after checking out the code.

It is good for the project to be set up, so that you can compile it. Then you can use IDE searching, navigation support, tool tips, etc, to orientate yourself much faster in the implementation. You can get a sanity check of the build environment by running JUnit's own unit tests (about 200 would likely fail out of 2000+, don't worry about that).

While studying the code, record new information you as learn it by enriching and revising the class model, and your list of constraints. Again it is not our point to reflect implementation classes one-to-one in your high-level model; rather to add information and to correct what was misunderstood.

[a] We mean a contributor to JUnit project, and not developers writing tests using JUnit in other projects.

Task size guide: You will end up with ca. 25-30 classes, including those added in the two prior exercises. Estimated time: about 2–3 hours, assuming that you are reasonably fluent with the modeling tool, can read Java code and documentation, and have used Git before.

Exercise A.7. After completing the three exercises above, reflect how modeling has supported the process of understanding the implementation of an unfamiliar system. Has it made the investigation more systematic? Has it made the comprehension easier? Have you ever referred to your models when trying to understand something in later phases? If you have worked in the group: did the models, and ability to draw support group discussions? Would the created model help, if you needed to explain what you understood to a colleague?

B Using the Eclipse Modeling Framework

B.1 Installing Eclipse Modeling Tools

Install a new copy of Eclipse by following the instructions below. Multiple instances of Eclipse in different versions can happily co-exist in your system. We recommend that you do not customize any existing installations of Eclipse on your PC. You would likely break the setup for your existing projects (if you already use Eclipse), and likely the resulting installation would be slightly incomplete, leading to incompatibilities (you do not want to waste time on them).

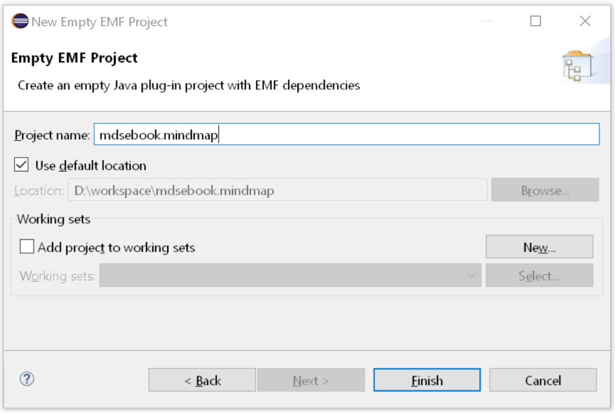
One way of handling multiple Eclipse installations is to have a global folder where you store all these installations, and then specific folders for each installation by choosing an appropriate name for it (e.g., a name that contains the year and release of Eclipse). In this folder, you extract the Eclipse archive and create a workspace folder where you should keep all your projects related to this installation (avoid opening the same workspace using different versions of Eclipse).

1. Make sure that at least Java JDK 8 (recommended since the release of Eclipse Neon2) is installed on your computer.
2. Download Eclipse Modeling Tools from https://www.eclipse.org/downloads/eclipse-packages.
3. Copy the downloaded ZIP archive to the folder you created for this Eclipse installation and decompress it.
4. Run Eclipse by double clicking the Eclipse icon in the newly created folder (or invoking the eclipse executable)

B.2 Create an EMF Project

Class diagramming in the Eclipse Modeling Framework (EMF) can be done using the Ecore language, which is a simple subset of UML class diagrams (cf. Sect. 3.3). We shall now accustom ourselves with the language and its associated tools. We will use our simple mind-map language from Sect. 3.7 as a running example.

Our objective is to show how to create a simple class model with EMF. In Eclipse, every file belongs to a project. Create a new project called `mdsebook.mindmap` via File → New → Project → Eclipse Modeling Framework → Empty EMF Project. See Fig. B.1.


New Empty EMF Project

Empty EMF Project

Create an empty Java plug-in project with EMF dependencies

Project name: mdsebook.mindmap

[✓] Use default location

Location: D:\workspace\mdsebook.mindmap

Working sets

[ ] Add project to working sets

Working sets:

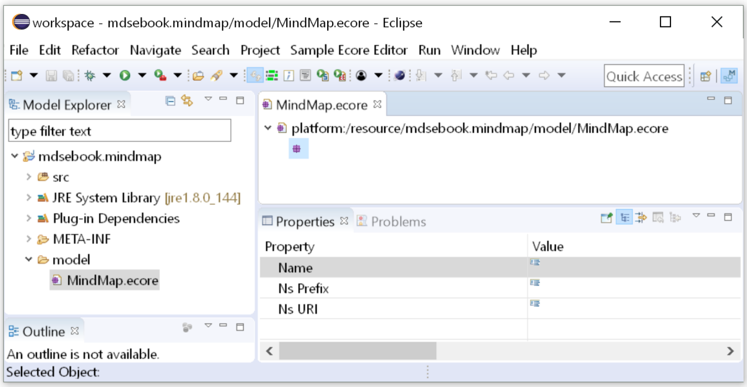
Figure B.1: Creating an EMF project

B.3 Create an Ecore Model (Meta-Model)

EMF distinguishes between model files (abstract syntax or the actual data) and diagram files (concrete syntax or the visualization of data). To create a model file select the model folder, right-click on it and select New → Other → Eclipse Modeling Framework → Ecore Model. Name the model `MindMap.ecore`. Ecore is the language in which we will be doing the class modeling, so `Mindmap.ecore` represents the meta-model of our language. Such Ecore models are stored in files with the `.ecore` extension.

B.4 Create an Ecore Diagram for the Ecore Model

In order to create a diagram file, right-click on the created Ecore file and select Initialize Ecore Diagram, then choose Entities in a Class Diagram and give the diagram a name (such as the default `class diagram`). Both the model and the diagram are initially empty. You should have two files in the *model* folder: `MindMap.ecore` and `MindMap.aird`. Open the former first.


workspace - mdsebook.mindmap/model/MindMap.ecore - Eclipse

type filter text

- mdsebook.mindmap
  - src
  - JRE System Library [jre1.8.0_144]
  - Plug-in Dependencies
  - META-INF
  - model
    - MindMap.ecore

platform:/resource/mdsebook.mindmap/model/MindMap.ecore

Property	Value
Name	
Ns Prefix	
Ns URI	

An outline is not available.

Selected Object:

Figure B.2: The **model** tree editor with the properties view in the bottom.

This will launch the Ecore tree editor, a generic editor that shows Ecore models as trees, as shown in Fig. B.2 (where the three is still empty, only the root node is shown).

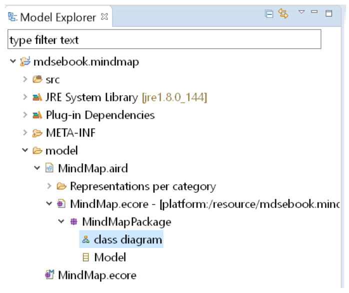
If not yet opened, open the Properties view via the menu Window → Show View → Other ... → General → Properties. This view allows you to modify the attributes of your model elements. Explore the available properties. At this point it is also useful to mention the Quick Access feature of Eclipse. Almost any deeply nested property or a menu item can be accessed quickly, if you remember its name. Press Ctrl+3 and type "properties", then select from the menu. This way a lot of complex operations can be speeded up.

Let us rename the unnamed package in the model editor (orange line in Fig. B.2). Let it be called `MindMapPackage`. This can be done in its Properties view, which is reachable, for instance, from the context menu in the model editor, or by double clicking on the node, if you closed it.

B.5 Class Modeling using the Ecore Diagram Editor

Classes. In order to create a new class in the model editor, select New Child from the context menu of the `MindMapPackage` node (named so above). Then select Eclass from the list of choices. A new node is created. You can name it in the Properties view. Please name it `Model`.

Observe that a class does not automatically appear in the diagram editor (see Fig. B.4). Note: The diagram editor can be opened by double-clicking the 'class diagram' entry in the hierarchy under the `WebApp.aird` file in the Model Explorer view, as shown in Fig. B.3. When the diagram is still empty, double click on "Double-click to initialize using the EPackage content." In our case, this will add the class `Model` created above.



```
Model Explorer
type filter text
mdsebook.mindmap
  src
  JRE System Library [jre1.8.0_144]
  Plug-in Dependencies
  META-INF
  model
    MindMap.aird
      Representations per category
      MindMap.ecore - [platform:/resource/mdsebook.minc
        MindMapPackage
          class diagram
          Model
    MindMap.ecore
```

Figure B.3: *Opening the* ***diagram editor*** *for the class diagram created for our Ecore model*

In the diagram editor, we create classes using the tool palette on the right side. Select Class and click anywhere on the canvas to create a class. Then rename it. Create classes for `MindMap`, `Topic`, and `NamedElement` (see Fig. B.5. Make `NamedElement` an abstract class by double-clicking it, which opens the properties, and selecting the "Abstract" checkbox. Also give `NamedElement` an attribute with the name `name` and the type `EString` (simply by hovering over the class, selecting the Attribute icon, and then just typing "name: EString" and pushing enter, which adds the attribute.

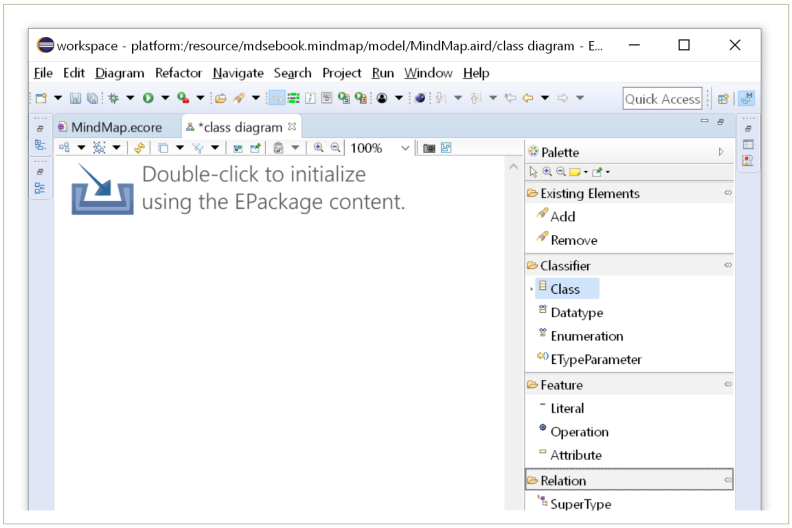

workspace - platform:/resource/mdsebook.mindmap/model/MindMap.aird/class diagram - E...
File Edit Diagram Refactor Navigate Search Project Run Window Help
Quick Access
MindMap.ecore
*class diagram
100%
Double-click to initialize using the EPackage content.
Palette
Existing Elements
Add
Remove
Classifier
Class
Datatype
Enumeration
ETypeParameter
Feature
Literal
Operation
Attribute
Relation
SuperType

Figure B.4: View of the **diagram** editor

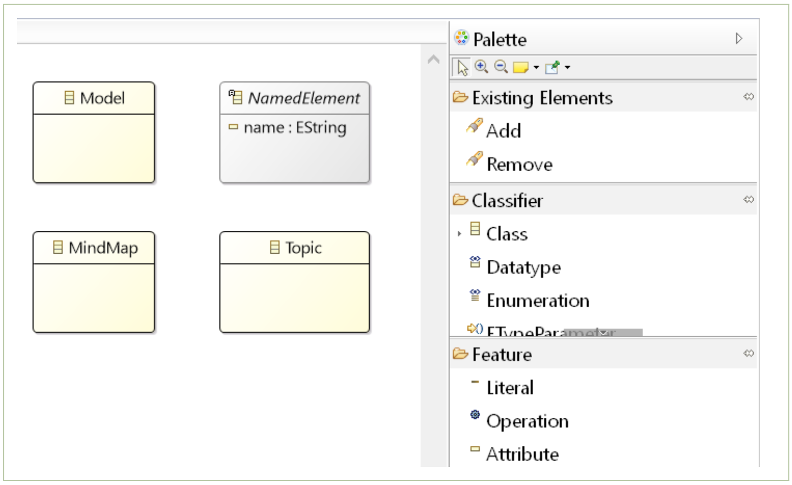


Figure B.5: An Ecore diagram with four classes

Observe that classes added to the diagram appear in the model (please check in the other editor).[1] *A diagram is a view on the model, so every element of the diagram must exist in the model, but not vice-versa.* Recall the class `Model` which we had created in the model and then added to the diagram by double-clicking “Double-click to initialize using the EPackage content.” If we had not done so, we could have added it to the diagram by choosing Existing Elements / Add from the palette.

Containment and Inheritance. Two kinds of relations are of special importance in class diagrams: generalization (inheritance, a *kind-of* relation) and

[1] Since different views are synchronized via files, you will only see the new classes if you save the diagram in the diagram editor. Always save the model (respectively the diagram) after changes when switching editors.

containment (a *part-of* relation). In the Ecore diagram editor, generalization is available from Relation/SuperType. Containment is available from the Relation/Composition palette. Add these two kinds of relations to your diagram: a `MindMap` is a part of a `Model`, `Topic` is a part of `MindMap`, `Topic` is also a part of itself (to realize sub topics), and both `MindMap` and `Topic` are a kind of `NamedElement` (i.e., these classes have a `name` attribute. Compare the result to Fig. B.6. Save the diagram, and observe the impact in the model editor. Observe that the compositions mindmap and topic are just a regular reference in the model, but their Containment property (see properties view) is set to true.

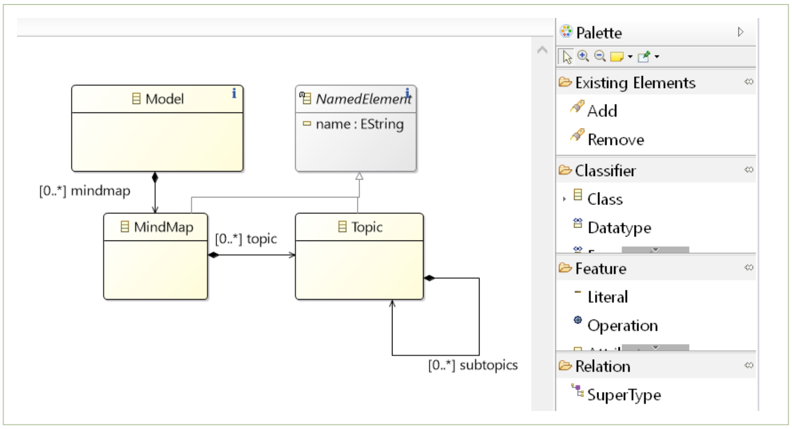


Figure B.6: *Composition and generalization. A black diamond is placed on the container side; an arrowhead next to the more general class.*

Other Elements. Experiment with adding attributes and references to your classes **both** in the diagram editor and in the model editor. Add at least the attributes author and description to the class `MindMap` and make sure they are of type `EString`. You can also define default values for attributes. We show an example of the default value “1” for an attribute editorVersion of the class `Model` in Fig. B.7. Try visualizing elements added in the model/tree editor in the class diagram, and locate elements added to the diagram in the model editor.

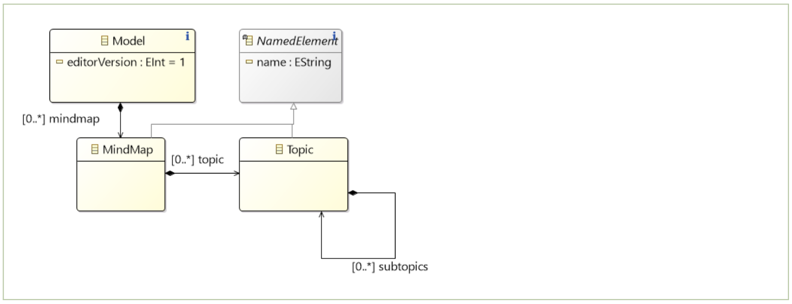


Figure B.7: *A default value defined for the attribute editorVersion*

Multiple Diagrams. Since diagrams are views on models, it is possible to create multiple diagrams for the same model. Try creating another diagram for your model, that only contains the classes `MindMap` and `Topic`. This can be done from the context menu of the `MindMapPackage` in the Project Explorer (or in the Model Explorer). Select New Representation and MindMapPackage class diagram. Existing classes can be added to this diagram as described above (Existing Elements / Add). The ability to create multiple diagrams for the same model is very useful when models become large. You can, for instance, create one view (diagram) showing the main part of your DSL design, and another one showing only types, or only the expression sub-language. It is quite common practice to create one diagram that only shows the generalization hierarchy (the *taxonomy* of concepts in the model), and the other one that shows the containment hierarchy (the *partonomy* of concepts in the model).

Class Duplication. A class with a given name can only exist once in each package (as each package defines a *namespace*). Try adding another class named `MindMap` to the model in the tree editor, then choose validate from the context menu of the package `MindMapPackage` (not just of the newly created class). You will receive an error message about a duplicate classifier. The same will happen if you try to add an existing class to a diagram in the diagram editor, using Classifier/Class, as opposed to Existing Elements/Add. This is equivalent to creating a duplicate class in the model editor. Please get rid of duplicate classes by renaming or deleting them (see below about deletion).

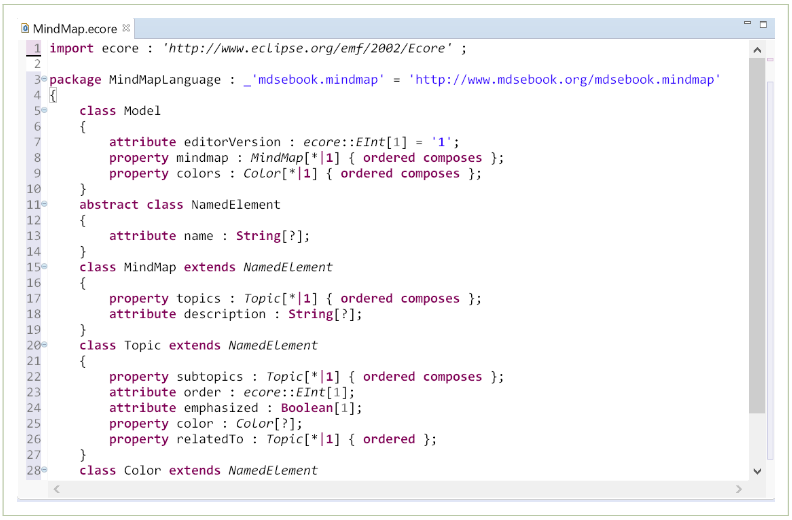
Let us also use this opportunity to fix the other two errors in the validation results of `MindMapPackage`. Set the property namespace prefix to, for instance, `mdsebook.mindmap` and the namespace URI to `http://www.mdsebook.org/mdseb`
The latter key will be used to load the models in your programs. It is important to always give different namespace URIs to your models. The framework gets confused and produces inexplicable errors when you try to load a model using a URI, and several model files in the workspace use the same one.

Deleting and Hiding. Try to delete an element in the model editor. It disappears from all diagrams of the same model (after saving). In the diagram editor, you have a choice:

- Hide simply hides the element (find it under Show/Hide in the context menu of the element in the diagram editor). The element still exists both in the model and in the diagram. You can make it visible again in the diagram by invoking Show/Hide menu.
- Delete from Diagram under Edit in the context menu, removes the element from the Diagram. Please test that after such removal the element remains in the model, but disappears from the diagram. It remains in all the other diagrams of the same model, which contained it before deletion.
- Delete from Model in the same context menu, permanently removes the element from the model, and consequently from all its diagrams (please verify).

B.6 Edit Ecore Models Using a Textual Syntax

It can be cumbersome to use the graphical Ecore editors, that is, the Ecore tree editor (Fig. B.2) or the Ecore diagram editor (e.g., Fig. B.4). Instead, there are Ecore editors that provide a textual syntax for Ecore that can be edited. Figure B.8 shows the OCLinEcore editor, whose primary purpose is to add OCL constraints into an Ecore model. However, it can also be used to solely edit the Ecore model through the textual syntax. The editor can be started by right-clicking the Ecore file and selecting Open With → OCLinEcore Editor. We found this editor a bit unstable, however. Using the editor might require installing additional OCL plugins into Eclipse, search for ’OCL’ in the Eclipse Marketplace.



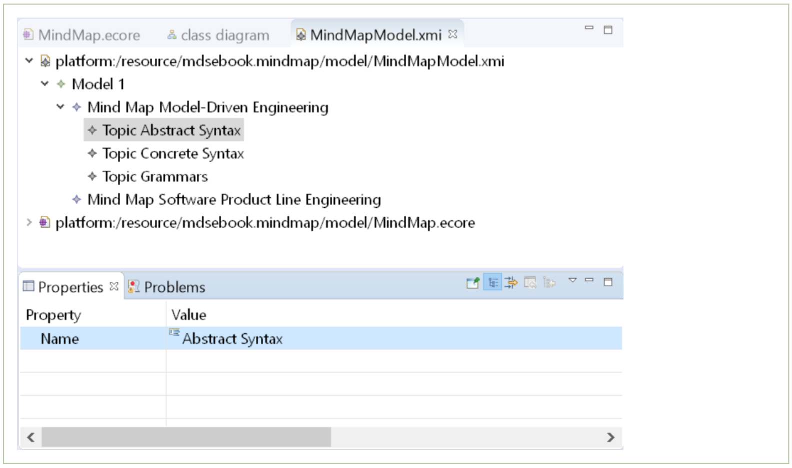
```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;

package MindMapLanguage : _'mdsebook.mindmap' = 'http://www.mdsebook.org/mdsebook.mindmap'
{
    class Model
    {
        attribute editorVersion : ecore::EInt[1] = '1';
        property mindmap : MindMap[*|1] { ordered composes };
        property colors : Color[*|1] { ordered composes };
    }
    abstract class NamedElement
    {
        attribute name : String[?];
    }
    class MindMap extends NamedElement
    {
        property topics : Topic[*|1] { ordered composes };
        attribute description : String[?];
    }
    class Topic extends NamedElement
    {
        property subtopics : Topic[*|1] { ordered composes };
        attribute order : ecore::EInt[1];
        attribute emphasized : Boolean[1];
        property color : Color[?];
        property relatedTo : Topic[*|1] { ordered };
    }
    class Color extends NamedElement
```

Figure B.8: OCLinEcore editor, used to edit Ecore files using a textual syntax

B.7 Create a Dynamic Instance

So far we have learned how to create class models and diagrams in Ecore. Next we will create an instance of this class diagram. Open the model `MindMap.ecore` in the package explorer or in the model editor. Select Create Dynamic Instance from the context menu of the `Model` class. In the file dialog, choose `MindMapModel.xmi` as the file name, since Ecore model instances (more precisely, in abstract syntax) are typically saved in the XMI format. A new tree editor is opened, containing a node of the instance tree (first line) and a reference to the Ecore model (second line).

Instances of class diagrams are *object diagrams*. If you open the first node, you will see that this diagram already contains one object of type `Model`. Add two child nodes to this node, both of type `MindMap`. You can give them names by filling in their name attribute in the properties view. Create three topics for the first mindmap. Compare to Fig. B.9. A duplication of types is possible, now. Obviously it is OK to have two mindmap objects in the model (while it is illegal to have two MindMap classes in the model). Note that changes to the instance model have no effect on the Ecore models and diagrams. This is because the creation of instances (data) does not change the types (models).



```
platform:/resource/mdsebook.mindmap/model/MindMapModel.xmi
  Model 1
    Mind Map Model-Driven Engineering
      Topic Abstract Syntax
      Topic Concrete Syntax
      Topic Grammars
    Mind Map Software Product Line Engineering
platform:/resource/mdsebook.mindmap/model/MindMap.ecore
```

Property	Value
Name	Abstract Syntax

Figure B.9: *Create Dynamic Instance*

B.8 Generate Language Infrastructure

Instead of creating a dynamic instance, you can also generate the language infrastructure (Model, Edit, and Editor Code) and then launch an editor to create an instance. The generation of code is controlled by a so-called genmodel, which configures the code generator. So, after you create the eCore model, you need to create a genmodel. You typically put the genmodel into the same folder as the Ecore model (e.g., into `model/`). Right-click on teh folder to open the context menu and then select New → Other → Eclipse Modeling Framework → EMF Generator Model. You can give it the same filename as the Ecore model, but with the extension `.genmodel`. Then select `Ecore model` as the model importer, and then select the respective Ecore file for which (usally via Browse Workspace) you want to generate code (here, `MindMap.ecore`). Then select Finish (the root package `MindMapLanguage` should be selected by default in this window).

Generate the editor by opening the genmodel file. On the root node, choose Generate All from the context menu. This will, among others, create a project called mdsebook.mindmap.editor. Launch the editor by choosing Run-As → Eclipse Application from the context menu of this editor project. Usually, some validation errors pop up, which you can ignore. This will launch an Eclipse instance that has quite many plugins (you could remove some not needed plugins in the launch configuration), but especially the generated editor plugins.

In the launched Eclipse instance, create a new project (choose just Project as the project type). In the project, create a new mindmap model by choosing, from the project's context menu, New → Other, and there select `MindMapLanguage Model`. In the following wizard, when asked about the Model Object, choose `Model`.

In the editor, you can now create an example mindmap. It is the same tree editor that launches when you edit a dynamic instance (see Appendix B.7 above). As an exercise, create at least seven topics, some of which should be nested. Use the Properties view (on any node, choose Show Properties View from the context menu) to set properties of each node. Recall that a node represents an instance of a class, a.k.a., object.

Further Reading

Whole books were written about Eclipse EMF. To understand EMF in its whole richness, read into the books of Steinberg et al. (2009), Budinsky et al. (2004) or Moore et al. (2004).

Lars Vogel also covers Eclipse EMF on his website http://www.vogella.de. A free tutorial on it for the current release of Eclipse can usually be found at http://www.vogella.de/articles/EclipseEMF/article.html.

C Xtext in a Nutshell

C.1 Syntax Overview

We will now explain the syntax definition using Xtext as an example, so that you can define concrete syntax yourself. We will rewrite the default grammar, generated by Xtext, into a simpler, more human-friendly one. A hands-on guide for using Xtext in Eclipse, with instructions on how to exactly generate the infrastructure and to run the generated editor is given in Appendix C.

The XText specification language is a variation of the familiar *Extended Backus-Naur Form* (EBNF) notation for context free grammars. EBNF is used by most parser generators, and it is included in curriculum of most compiler courses.

We shall now discuss the main syntactic elements of the Xtext language, by presenting a simple grammar for an artificial trip language, whose meta-model is shown in Fig. C.1. The grammar specification starts with the name (in the same format as fully qualified Class name in Java): `grammar org.xtext.example.mydsl.MyDsl`.

In order to enable reuse in language definition, Xtext allows importing other grammars. In our example above, we have included the grammar that describes standard terminals in a typical programming language:

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals
```

Then we import the meta-models used as the abstract syntax:

```
import "http://URI.declared.in.the/metamodel"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

The first model imported is the model presented in Figure C.1. We also import Ecore itself, in order to be able to use its types, for instance EString. Note that the elements of trip become available in the default name space (no *as* clause). So later in the grammar, types like Person are referred to directly. At the same time the Ecore types are prefixed by the name space `ecore`. Double colon is used as the name space prefix operator.

The first symbol used in the grammar specification (the left hand side of the first production) is the start symbol of the grammar. Here: `TripModel`. The `returns` construct allows to specify the type representing a nonterminal in the abstract syntax. The default type has the same name as the nonterminal, so most of the *returns* clauses in the generated grammar are redundant.

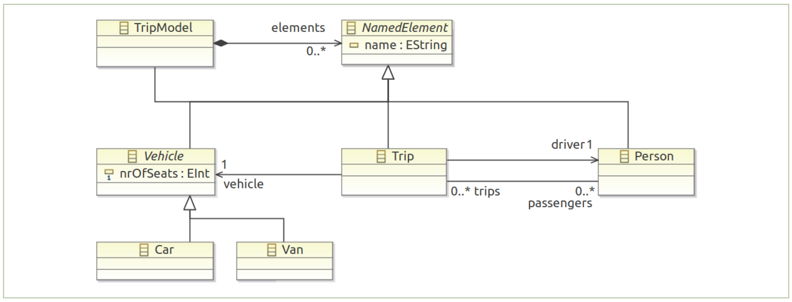


Figure C.1: Abstract syntax (the meta-model) of the trip *language*

Terminals are simply introduced as string literals. For example `'Car'` and the braces in the following rule:

```
Car: 'Car' name=EString '{' 'nrOfSeats' nrOfSeats=EInt '}';
```

A value resulting from parsing a nonterminal can be stored directly in a property of the current abstract syntax object. For example, the above production says that an object of type Car will be constructed upon its successful application. The name of the car (`Car.getName()` in Java) will be initialized with the value of a string directly following the first keyword. Similarly the number of seats will be initialized to an integer value following slightly later. In general a property of any time (including class types) can be assigned with an object constructed by invoked productions.

In the following rule we see how elements parsed can be used to populate collections:

```
Person returns Person:
  'Person' name=EString
  '{'
    ('trips' '(' trips+=[Trip|EString] ( "," trips+=[Trip|EString])* ')' )?
  '}';
```

If a property of an abstract syntax object is a collection, we can add a value to the collection (as opposed to replacing the collection) using the `+=` operator instead of assignment. This happens for both vehicles and elements above. There is no null pointer error, since the collections are initialized to be empty upon object creation (by EMF).

The braced type name (`{TripModel}`) enforces creating an instance of a given type. This is useful if we are parsing a concept that is represented by an abstract type with several possible implementations. For example, parsing named elements, could be more concisely written without introducing nonterminals for Cars and Persons as:

```
NamedElement returns NamedElement:
      Trip
    | Car
    | TripModel
    | Van
    | {Person} 'Person' name=EString '{'
        ('trips' '(' trips+=[Trip|EString]
                  ( "," trips+=[Trip|EString])* ')' )?
'}';
```

The use of this construct in this example is redundant. It is only there because the grammar is automatically generated and needs to also cater for other complex meta-models.

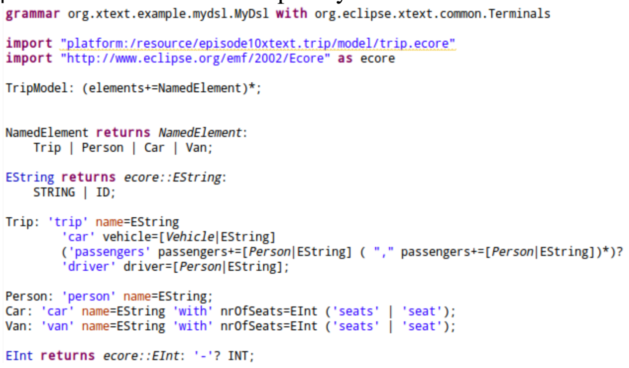
Remaining syntax: alternatives (|), repetition (+,*), optionality (?) is familiar from most regular expression dialects, and the meaning is as expected.

A crucial ability of Xtext is resolving references. The syntax for a cross-reference is `[TypeName|RuleCall]`, where `RuleCall` defaults to `ID` if omitted. The parser only parses the name of the cross-referenced element using the ID rule and stores it internally. Later on, the linker establishes the cross-reference using the name, the defined cross-reference's type (Entity in this case) and the defined scoping rules. For example:

```
Person: 'Person' name=EString '\{'
    ('trips' '(' trips+=[Trip|EString]
              ( "," trips+=[Trip|EString])* ')' )?
'\}';
```

There is much more to Xtext than we present, but this is sufficient for creating simple languages. Among other elements, of the highest interest are probably customizable scoping semantics (what names are visible in what scopes), and fully qualified name support for references across name spaces/scopes. These are described in the Xtext documentation.

Finally, we present another version of the grammar for the same language Trip that leads to a much simpler syntax:



```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/episode10xtext.trip/model/trip.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TripModel: (elements+=NamedElement)*;


NamedElement returns NamedElement:
    Trip | Person | Car | Van;

EString returns ecore::EString:
    STRING | ID;

Trip: 'trip' name=EString
        'car' vehicle=[Vehicle|EString]
        ('passengers' passengers+=[Person|EString] ( "," passengers+=[Person|EString])*)?
        'driver' driver=[Person|EString];

Person: 'person' name=EString;
Car: 'car' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');
Van: 'van' name=EString 'with' nrOfSeats=EInt ('seats' | 'seat');

EInt returns ecore::EInt: '-'? INT;
```

It takes about minutes, not days, to rewrite the default generated syntax specification to this one, including generating a new Xtext-based editor for models. An example model in this syntax looks as follows:

```
person Andrzej
person Helge
person Thorsten
person Joachim

car VWPolo with 4 seats
car Trabant with 4 seats

trip MDSETrip
    car VWPolo
    passengers  Helge, Andrzej, Thorsten, Joachim
    driver Joachim
```

Since Xtext registers suitable handlers, if you insist that Eclipse should open a .trip file in an ecore-like model editor, it will automatically involve the parser and open the editor, just as if the mode was stored in an xmi file.

Finally, C-like comments (both block comments, and line comments) are automatically supported in languages built with Xtext.

C.2 Creating DSLs with Xtext

Xtext is a popular language workbench for Eclipse. It supports the design and development of textual languages. The project's documentation provides the following concise characterization: "*Xtext provides you with a set of domain-specific languages and modern APIs to describe the different aspects of your programming language. Based on that information it gives you a full implementation of that language running on the JVM. The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows you to use Xtext together with other EMF frameworks and tools (...). In addition to this nice runtime architecture, you will get a full-blown Eclipse IDE specifically tailored for your language.*" (XText online documentation)

We give a brief guide to Xtext by showing how to automatically derive a concrete textual syntax for one of the examples used before in this book. Xtext supports both the grammar-first and the meta-model first ways of designing DSLs (see box "Grammar-First or Model-First?" in Sect. 3.6). In the former case, it automatically generates the meta-model from the grammar, while in the latter case, it automatically generates a crude grammar from a meta-model. We use the Ecore model of Fig. 3.1 as the meta-model for this guide (see page 58).

We follow these steps:

1. Create the .ecore file with the meta-model. Ensure that the meta-model has a single partonomy. Using nested packages is discouraged. Use one top-level package. Add names to as many model elements as possible, of course, if it makes sense. The name property is used with default editors and for default name/reference resolution. Also ensure that the model has a unique URI in the workspace (otherwise the framework will have difficulty identifying it).

2. Create the default generator model for the fsm.ecore file (File / New / Other ... / EMF Generator Model).

3. Generate the model code from this new genmodel (context menu of the fsm in the .genmodel editor)

4. Add the *Xtext nature* to the project containing your fsm.ecore meta-model (context menu of the project folder / Configure / Convert to Xtext Project).
   Otherwise, the import of the meta-model via the grammar's (fsm.xtext) import statement will not work (line 4 in Fig. C.2). An alternative to this conversion is to use "platform:/resource/mdsebook.fsm/model/fsm.ecore" as the import string in the grammar (fsm.xtext), instead of the meta-model URI; which causes a warning, however.
5. Create a new Xtext project from existing ecore models (using a dedicated Project creation wizard in Eclipse). Choose ecore.genmodel and the `Model` class as the root element (in the "Entry rule" dropdown menu). Choose `.fsm` as the file extension for this language.
6. A new project is created, and the default grammar specification for your language generated and opened in a text editor (the `fsm.xtext` file). This generated (and clumsy) syntax is shown in Fig. C.2.
7. Generate Xtext artifacts (in the grammar editor use the context menu command: Run As -> Generate Xtext Artifacts). Several projects are created.
8. Launch the main Xtext project as an Eclipse application ('Run as'). A new instance of Eclipse should start. Create a new project and a file with the .fsm extension, to open your newly generated editor. Experiment with syntax highlighting, code completion, name resolution, and interactive error reporting.

Fig. C.3 shows a screenshot of the running editor generated by Xtext. The terminals of the above grammar specification have been directly turned into keywords in the editor. The strings are referring by name to other objects (so the two occurrences of `"initial"` are actually linked in the constructed AST).

The model in Fig. C.3 is correct. However, if it contained static syntactic errors (like dangling references to states, or syntax errors), they would be highlighted in the editor, and listed in the Problem View of the Eclipe Workbench. The editor not only parses the model as you type, but also performs static validation of references. It also does type checking, and can be integrated with other static semantics constraints.

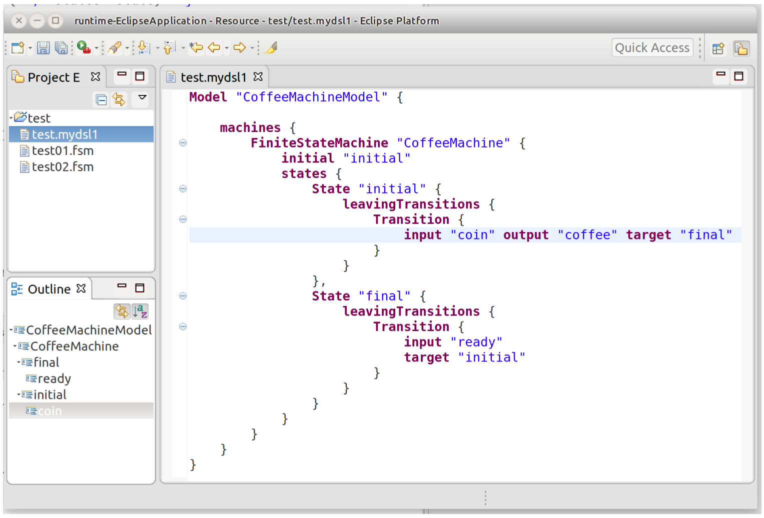
The newest versions of the Xtext framework can also generate editors as plugins for JetBrain's IntelliJ, and web-based editors (JavaScript-based) from the very same input specifications. These editors also produce instances conforming to the same Ecore meta-model. Moreover, Xtext generates build setups for Maven and Gradle, to allow automatically building the projects outside Eclipse.

```
\textbf{enum} SimpleTypeEnum:
BOOLEAN='boolean' | OBJECT='Object' | OBJECT_ARRAY='Object[]' |
DOUBLE="double" | LONG="long" | BYTE_ARRAY="byte[]" |
SHORT = "short" | INT ="int" | FLOAT = "float";
```

```
Exp \textbf{returns} Exp:
  Term ( \{BOp.lexpr=\textbf{current}\} operator='+' rexpr=Exp)*;
```

```
grammar org.xtext.example.mydsl1.MyDsl
  with org.eclipse.xtext.common.Terminals

import "http://www.mdsebook.org/mdsebook.fsm"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model returns Model:
    {Model}
    'Model'
    name=EString
    '{'
        ('machines' '{' machines+=FiniteStateMachine ( ","
      machines+=FiniteStateMachine)* '}' )?
    '}';

EString returns ecore::EString:
    STRING | ID;

FiniteStateMachine returns FiniteStateMachine:
    'FiniteStateMachine'
    name=EString
    '{'
        'initial' initial=[State|EString]
        'states' '{' states+=State ( "," states+=State)* '}'
    '}';

State returns State:
    {State}
    'State'
    name=EString
    '{'
        ('leavingTransitions' '{'
        leavingTransitions+=Transition
        ( "," leavingTransitions+=Transition)* '}' )?
    '}';

Transition returns Transition:
    'Transition'
    '{'
        'input' input=EString
        ('output' output=EString)?
        'target' target=[State|EString]
    '}';
```

Figure C.2: *The concrete syntax definition automatically derived from the FSM meta-model of Fig. 3.1.*



```
Model "CoffeeMachineModel" {

    machines {
        FiniteStateMachine "CoffeeMachine" {
            initial "initial"
            states {
                State "initial" {
                    leavingTransitions {
                        Transition {
                            input "coin" output "coffee" target "final"
                        }
                    }
                },
                State "final" {
                    leavingTransitions {
                        Transition {
                            input "ready"
                            target "initial"
                        }
                    }
                }
            }
        }
    }
}
```

Figure C.3: *The generated editor with default concrete syntax of Fig. C.2. The bottom left pane shows a sketchy abstract syntax tree for the model ni the right pane.*