### CHAPTER 10

## DSM LANGUAGE DEFINITION

Defining a language for development work is usually thought to be a difficult task. This may certainly be true when building a language for everyone and for every system. How could you create a modeling language if you did not know its intended purpose, users and applications to be modeled?

The language definition task becomes considerably easier when the language need only work for one problem domain in one company. You can focus on a restricted domain, the area of your interest. This is often the same area where you have already been working. There are thus most likely some fundamental concepts already available and in use, even if they are only used in spoken language, product presentation slides, high-level product designs, or requirements. Actually, most people already have a domain-specific vocabulary in use and that vocabulary exists with good reason: it is relevant when discussing such systems. Usually, there is no need to introduce a whole new domain-specific language as it is already in use, albeit implicitly and partially. The person who specifies a Domain-Specific Modeling (DSM) language identifies the modeling concepts in detail and formalizes them by creating a metamodel.

10.1 INTRODUCTION AND OBJECTIVES

We describe in this chapter how to define one or several integrated modeling languages for a specific problem domain: from initial language concept identification to testing and maintenance. Throughout the chapter, we use the examples from Part III to

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen Copyright # 2008 John Wiley & Sons, Inc.

227

illustrate language definition in detail. First, in Section 10.2 we look at how to find and define the modeling concepts and their characteristics. This leads us to introduce language specification languages: Section 10.3 describes metamodeling as a technique for language definition and formalization. Having a formalization mechanism, our next concern in Section 10.4 is identifying and specifying various connections and rules among the modeling concepts. We then extend the scope from one language to language integration: having multiple languages that focus on different aspects of the problem domain. In Section 10.6, we give guidelines for defining concrete syntax: the notation of modeling languages. Metamodels provide an extra benefit for language definition as they can be instantiated to prototype and test the language. Section 10.7 gives guidelines for language testing, both before and after the language has been used. Finally, a DSM language almost always evolves along with the domain and our understanding of it. Section 10.8 describes language refinement and space and gives guidelines for maintaining the language already in use.

10.2 IDENTIFYING AND DEFINING MODELING CONCEPTS

Language defines the boundary to our world: it sets what we can describe and also what we can’t (Wittgenstein, 1922). For DSM the latter is crucial, as narrowing down the design space makes it possible to have generators that target full code generation. If it is not possible to narrow down the scope, then most likely the modeling language is unusable for generating the required code or other artifacts. In other words, we have created another general-purpose modeling language. The first rule of language definition is, therefore, to start with the better understood parts of the domain and extend the modeling language gradually to cover the full domain—possibly integrating with other languages. It is worth remembering that DSM languages should never be considered to permanently restrict our view of the world because they can be changed when needed. If the domain changes or our understanding of the domain changes, we should be able to change the language too.

The language creator needs to find concepts and abstractions that are relevant for a given development situation. Generally, the main domain concepts map to the main modeling concepts, while others will be captured as properties, connections, submodels, or links to models in other languages. While defining the languages, the balancewith thegenerator, domain framework, and available components also needs to be decided. Next, we describe approaches to identify modeling concepts and map them to some of the models of computation behind every modeling language. Although individual modeling concepts are most often applied with one language only, models are often integrated and reused. The language creator can also identify connections between modeling concepts and inspect language use situations where existing model elements could usefully be reused. Unlike in a general-purpose language, a domainspecific language can enforce reuse within the domain and address model integration where different aspects of the domain are combined into one language or several integrated languages. In the latter case, the parts of the metamodel are related by reusing the same elements or linking elements between different languages.

The identification of the modeling concepts is normally highly iterative and may need to be repeated several times. You definitely can’t identify all the concepts immediately and therefore it helps to define languages in stages and try them out by making example models. The number of iterations depends on many factors: the size of the domain, its stability, availability of domain knowledge, and the experience of the language definers.

10.2.1 Where to Find Modeling Concepts

Although it may be tempting to use concepts that originate from the code as a starting point for language definition, higher abstraction, and thus better productivity can be achieved if the modeling concepts are from the nonimplementation concepts. A study analyzing DSM language creation approaches in over 20 industry cases (Tolvanen and Kelly, 2005) shows that concepts coming from the look and feel of the system built, the variability space, and the domain expert’s concepts lead to higher abstraction than those originating from generation output. In other words, the language definer should focus more on the problem domain than its code (the solution domain in Jackson, 1995). Naturally, while aspiring toward a higher level of abstraction, we need to keep in mind that ultimately we need to have the ability to generate code from models too. Starting from domain concepts is always better, though: adding coding concepts later on is usually easier than vice versa.

Problem domain concepts also have other characteristics that make them good candidates for the language:

. They are usually already known and used. They can be found while communicating with customers but also among development teams. It is not necessary for the languages to introduce new concepts: they can build on existing ones when possible. Using concepts that are already in use is also relevant for the acceptance of DSM. People are comfortable with the concepts, and they don’t need to invest in learning new languages. In addition to concepts, it is often a good approach to base the notation of the language on problem domain symbols already in use (see Section 10.6).

. Problem domain concepts usually have established semantics in place. The language definer can then base the language on existing definitions and, if they are not fully available, can ask domain experts. For example, in the domain of mobile User Interface (UI) applications, developers know the concept of the soft key: two or more keys whose functions change based on the application context. Similarly, in the banking domain, everybody knows what a bank day means. If the semantics of a particular concept cannot be defined, most likely it is not a relevant concept for a language either.

. They establish a natural mapping to the actual problem being addressed by DSM. This makes model creation easier in the first place and enables modeling operations like reusing model elements at the domain level. A close mapping to the domain also makes models easier to read, understand, remember, check, and communicate with.

Not all problem domain concepts are suitable language concepts. Concepts that every feature, application, or product always has should not normally be part of the language. Why would we want to model something that is always the same? Instead they should be provided via components and code libraries or produced by the generator. In modeling languages, only those concepts that allow describing variation should be considered. Perhaps it is easiest to consider variation as a property of the language element. For example, buttons in a digital wristwatch (Chapter 9) need to be differentiated and thus the button concept has a property to specify its name. Variation can also be expressed by connecting language elements with each other. For instance, in the watch case an action does not have any properties of its own, but the action modeling concept is needed to specify how time variables are modified, alarms used, and icons shown and hidden.

Concepts that can be identified from the combination of existing concepts should also usually be excluded. For example, in the mobile phone case (Chapter 8), there is no explicit button concept that the user presses to indicate navigation paths in the application. Unlike in the watch example, where the number of buttons and their labels can vary, in the mobile phone case they are implicitly presented with the different navigation relationships: one for default selection, one for cancelling, and one for accessing menus. As alternative navigation paths need to be specified anyway, a special button concept in the language would be unnecessary. Keeping the language smaller is better, as then it becomes easier to learn and use.

Sources for Modeling Concepts Identification of the relevant concepts in the language is largely dependent on creative insights and experience in the domain. It helps if one has been involved in making similar kinds of applications in the past. Whenever possible, you should consult people more experienced in the domain, such as problem domain experts, architects, and lead developers. Their insights and opinions are often the main source for creating the DSM solution. You can interview them, observe their work routines, and use other mechanisms that reveal problem domain characteristics. Keep in mind, however, that the person specifying the language into a metamodel and implementing the language into a tool does not necessarily need to be experienced in the domain.

The candidate concepts for the modeling language can be found in very different sources. We can identify some of them from the jargon and vocabulary in use. Frequently used concepts exist with good reason: people find them relevant and concise when discussing a product and its features. The vocabulary often provides the best starting point, as it mostly uses natural concepts: people do not think of solutions immediately in coding terms. Starting from the existing vocabulary also means that there is no need to introduce a new, unfamiliar set of terms or map existing concepts and their semantics to those provided by some external languages. What does the Unified Modeling Language (UML) (or SysML, IDEF, BPMN, etc.) know, for instance, about banking applications, pacemakers, or applications you are developing? It is far better to use the concepts of your domain in a language than map them to external concepts and related semantics.

Other typical sources for finding candidate concepts from the domain include the following:

. Architecture: A description of the architecture is often a good source since the architecture usually operates on domain concepts. This is especially true for embedded systems. Usually, the architecture is also the most stable part and reveals core abstractions about the application elements and their behavior.

. Existing products, applications, features, and related manuals: These capture the structure, behavior, and semantics, but unfortunately their complete analysis is not always practical as it may be too time consuming to do an exhaustive analysis of all the products. It is more convenient to select a representative set of applications for more detailed inspection. Naturally, those kinds of applications should be selected that have functionality closest to the newly planned ones. It can also be that the applications and their features have not yet been developed. Then, you may look into similar kinds of applications in the same domain or inspect earlier generations or versions.

. Available specifications: By analyzing existing descriptions of the features, applications, or products, you can understand the structure of the domain and translate it to a conceptual schema. Requirement documents are especially good to raising the level of abstraction as they usually focus on the problem domain rather than the implementation. Requirements also map the customer terminology, making wider DSM use possible: customers can then better read and check the models. The specifications don’t need to be particularly formal or be based on some known specification language. Actually, models made without any restrictions lead modelers to capture the domain in a manner they see as most effective. Inspecting how they want to approach the problem and using alternative views immediately reveal the approach they see as most “natural.” In this sense, you should never underestimate presentation slides, drawing tools, or whiteboards. They are great for making specifications that map closer to the problem domain—but often poor for any automation like reuse, checking, sharing, analysis, generation, and so on.

. Patterns: If a company has an established collection of patterns, they may describe domain concepts or reveal common structures within the problem domain. Domain-specific patterns may also be available elsewhere that are representative of the problem domain under examination. Here, pattern matching can be applied to check the ingredients of a pattern and pattern recognition to detect underlying patterns.

. Target environment and its interfaces: Existing libraries, component frameworks, and interfaces were often made to raise the level of abstraction—but with limited guidance and automation for their use. Inspecting them shows how the applications and features are to be built and which services are already available.

. Code: Abstractions can be identified with a bottom-up approach: examine existing applications and features and generalize from them. This is not limited to finding abstractions for the language, but also helps identify the structures of generatedcode(seeChapter11fordetails).Inparticular,theadviceofexperienced programmers and coding guidelines need to be acknowledged: other developers follow these as they relate closely to established manual practices. It is not recommended, however, to follow only a bottom-up approach as you also need to predict future possibilities. This is important since after automating the development tasks with DSM you can develop features or whole applications that you could not develop earlier because of the higher costs or longer development times.

Sometimes, the above-mentioned sources may not be available: the domain is new, there are no past experiences available, or the specifications were not made available in the first place. The domain knowledge is then scattered in the organization and found only in individuals’ memory. Then, you should create concrete examples of alternative ways to specify the problem—not just one but multiple examples with prototypes. The sample applications are not only useful for identifying abstractions but can also be used later to test the DSM solution.

10.2.2 Useful Categories of Modeling Concept Sources

Domain concepts often resemble each other since many of them originate from the same source. Finding one good candidate often leads to finding other similar ones. The sources we have found useful for finding language concepts can be categorized as follows:

. Physical product structure

. Look and feel of the system

. Variability space

. Domain (expert) concepts

. Generation output

In practice, you often need to look at more than one category, but each offers a clear strategy with which start. We will discuss each category in more detail and provide some examples.

Physical Structure Physical structures usually provide a good starting point for the language definition, as they are relatively easy to identify and clearly restricted. For example, a language for developing automation systems for a paper factory or power plant could be based on problem domain concepts like valves, motors, sensors, and controls. A valve will have attributes like size and direction and rules on how it may relate to motors and sensors. By analyzing the physical structure of the product, other candidate concepts could be identified. Although a language may specify only software, many aspects of the software are closely connected to hardware. For example, the language concept valve needs to represent not the actual device or hardware but its controller or interface.

Typical problem domain areas where languages are partly based on physical structure can be found, for example, from communication systems, network-related

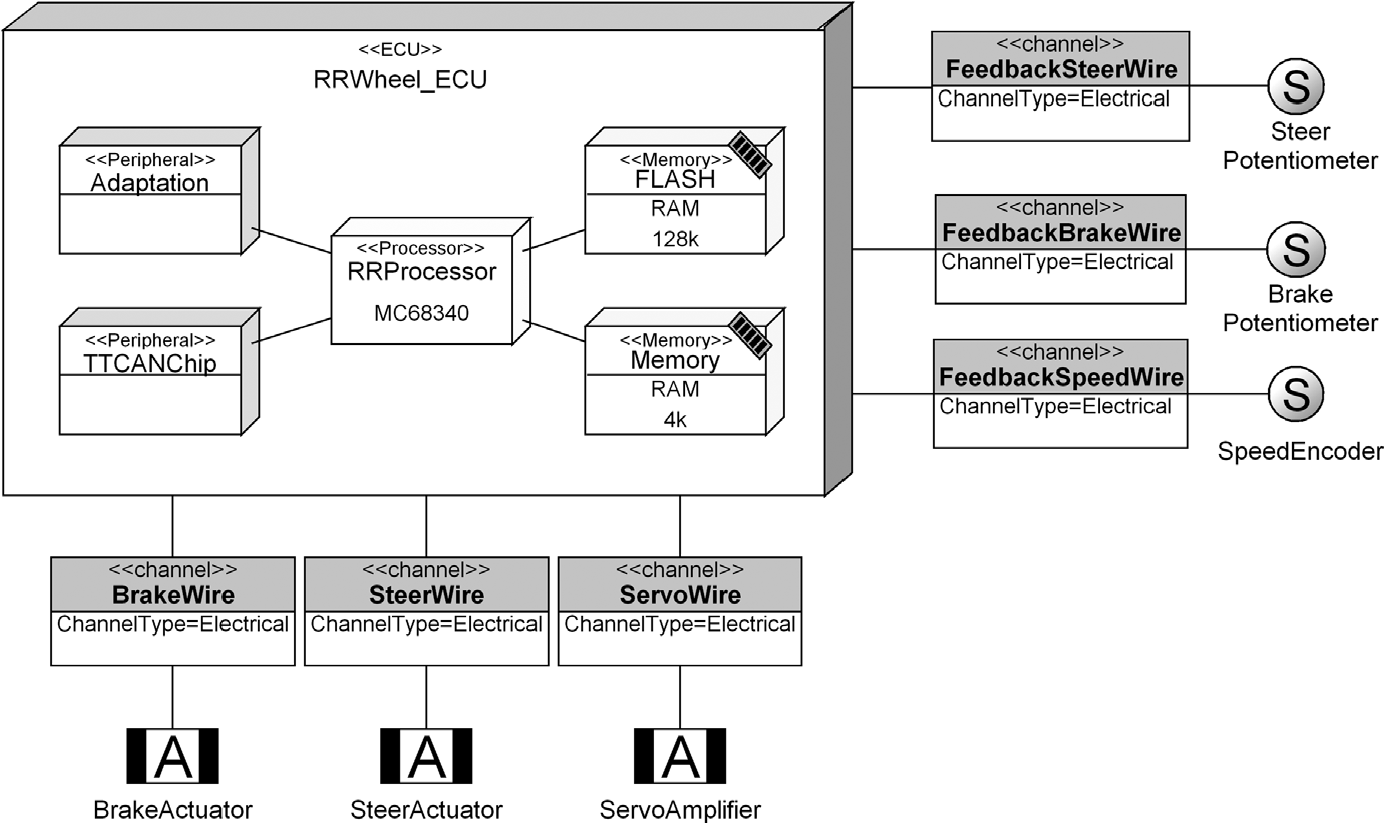


FIGURE 10.1 Physical structure-based language for modeling the hardware architecture in automobiles (based on EAST-ADL)

software, industrial automation, railway control, power and electricity control, home automation, logistic systems, and hardware architecture. Also, the design of distributed systems usually needs a language that is at least partially based on the concepts found from physical structures.

Figure 10.1 illustrates a DSM language that uses physical entities as modeling concepts. In EAST-ADL, one of its five modeling languages focuses on describing the hardware architecture in automobiles. Shown here is the architecture for electronic control units (ECU) with processors and memories connected through a CAN bus. The language provides several alternatives for bus types and constraints on the application of buses in the described hardware architecture. These modeling concepts can be identified by analyzing all the possible hardware components a car may have.

Languages based on physical structures usually focus on static declarative modes but may also include behavioral elements. Designs in such a language usually provide configuration data for the rest of the generation process and are usually linked to other models in order to achieve more comprehensive code generation.

Look and Feel of the System Products whose design can be understood by seeing, touching, or hearing (HMI/MMI systems) often lead to languages that apply end-user product concepts as modeling concepts. A language for defining voice menus can include concepts like “menu,” “prompt,” and “voice entry” as well as guidelines on how these may be linked to achieve user navigation. This type of language is quite easy to define and test, as it has “visible” counterparts in the actual product.

The language developer can thus look for language concepts by analyzing how the user of the product uses it. The product manuals and user guides that explain features, give guidelines for applying functions, or help in navigating the system all provide good sources for candidate modeling concepts. These kinds of languages are typical when targeting application development for Personal Digital Assistants (PDAs), mobile phones, diving instruments, wrist computers, and other consumer electronics as well as automotive infotainment and navigation systems. Perhaps one of the biggest areas in software development for these languages is the GUI navigation of typical administrative and enterprise applications.

Figure 10.2 shows an application design in a language that is clearly based on look and feel. The language targets the human-machine interface of an automotive infotainment system, specifying displays and their content together with the behavioral logic of the applications. If you are familiar with some infotainment applications, like navigation or setting preferences for traffic announcements, then you will most likely understand what the application does just by looking at the model.

Look and feel are represented directly in the language by using as modeling concepts the actual displays, user interface widgets, and user controls provided by the system. Having identified modeling concepts for one kind of display, widget type, or navigation path, the other modeling concepts can be defined similarly. It is relevant to note that, although the underlying framework and way to map these modeling concepts to individual code may differ among the concepts, it does not matter in the language creation phase. Basing a language purely on look and feel is not normally enough: You also need to identify and map non-GUI concepts to other concepts in the language or find a mapping to other languages.

Variability Space One efficient approach to start defining a language is to focus on variability: you define the language so that variability options are captured by the modeling concepts, and the modeler’s role is to concentrate on the areas that differ

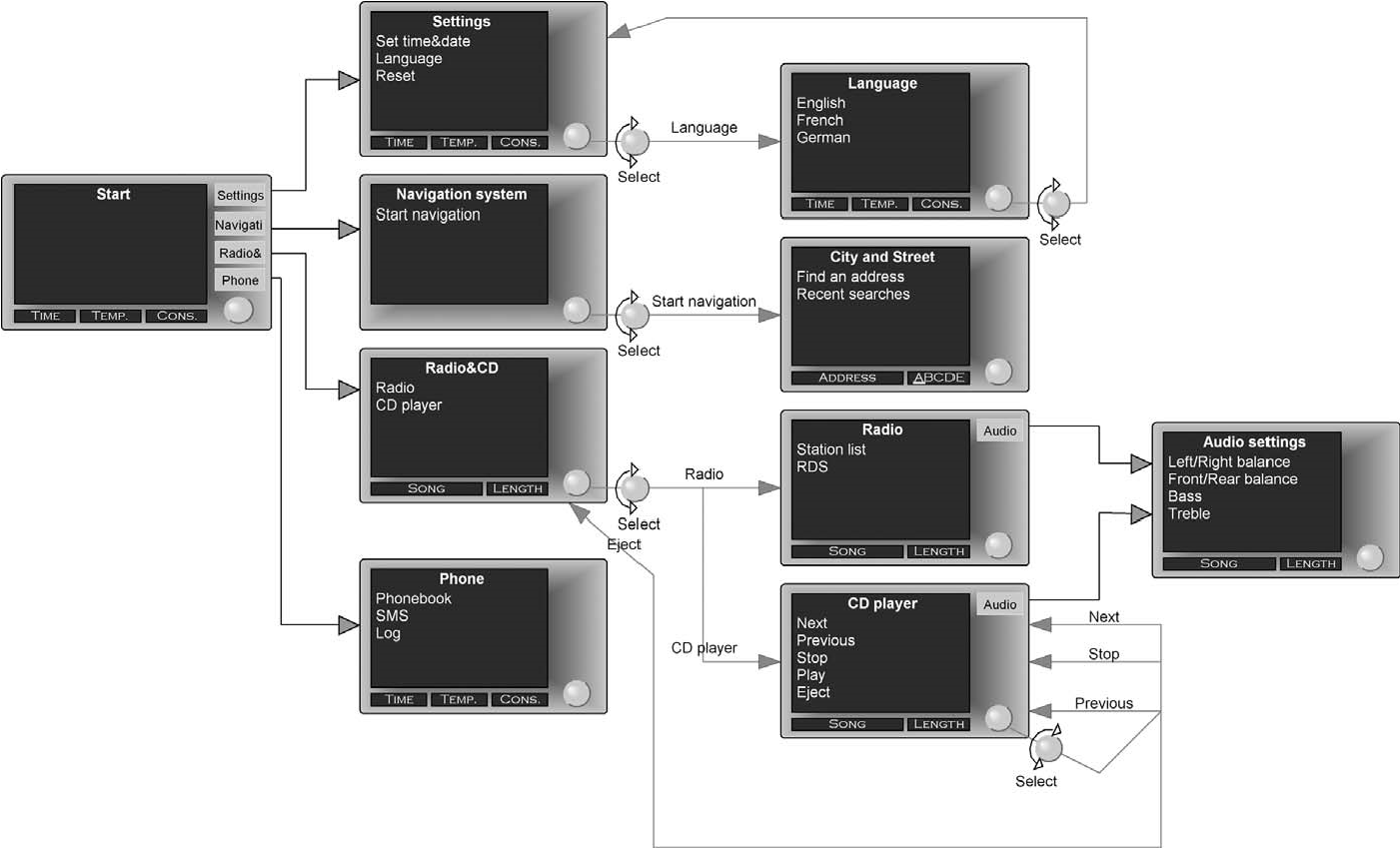


FIGURE 10.2 Language based on the look and feel of a car infotainment system

between different products or features. Long term success in defining a language here depends largely on your capability to predict what kind of variation space is needed in future products. Note that although this kind of language can most often be found in product-line development, it does not necessitate having multiple products: Variation also exists among features of a single product.

Language definition based on variability comes down to conducting a thorough domain analysis (Weiss and Lai, 1999): Identifying which abstractions are the same for all applications and which are different. Static variation is usually easy to cope with—developers have been making parameter tables and wizards to choose among alternatives for decades. Things get more complicated when the parameter choices depend on other parameter choices and here feature modeling (Kang et al., 1990) is useful and often applied along with configuration tools. Note that variation at the code level is not so relevant here since a generator may produce the required code to one or more places in various files to implement the variability. Parameter and feature choice approaches usually break down if we also want to tackle variability that is of a dynamic nature, or if we want to create new features and functionality inside the current variation space. This is a common situation since companies have hardly ever made all the product features to be selected.

Figure 10.3 illustrates the spectrum of variability. Wizards and feature-based configuration focus on making choices among known decisions and features. Domain-specific languages do not set choices explicitly but give a practically infinite space to set variation. You do not know all variants, as they can be numerous. There are as many variants as there are ways to instantiate the metamodel.

In the simplest case of modeling language design, the variability can be represented solely as properties of modeling objects—something similar to parameter

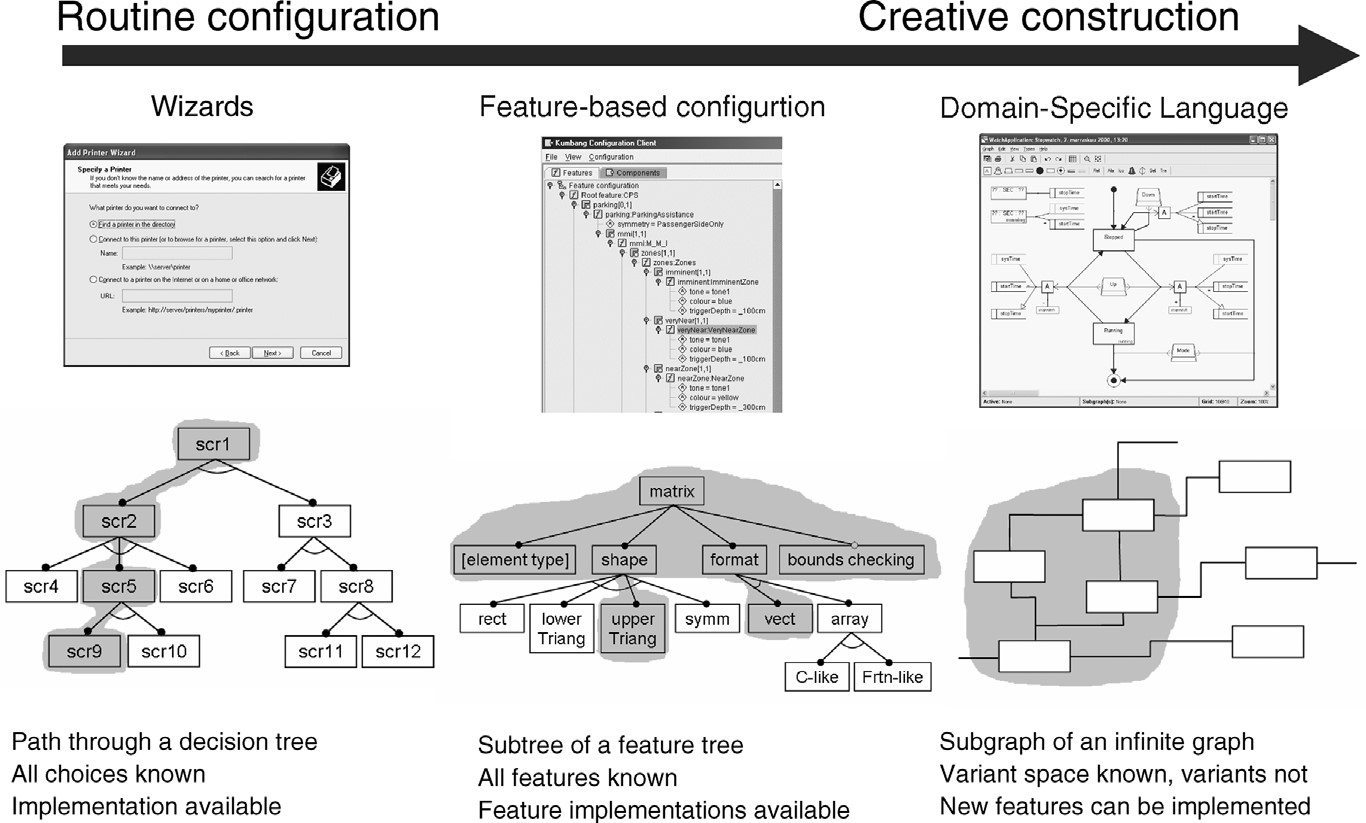


FIGURE 10.3 Spectrum of variability (modified from Czarnecki, 2004)

lists. The possible values a given property may have can be defined as a list containing the predefined legal values of that variability point. In a little more complex cases, and similar to feature models, variability can be expressed via connections between modeling elements, their linkages to submodels, and so on. Language concepts may also be rich enough to allow creating totally new implementations in the variation space set by the language. In this way, all the variants do not need to be made yet, or even thought about, since the creation of new variants can be done by creating totally new specifications. Among the examples in Part III, the representative case for this is the digital wristwatch language (see Chapter 9).

Domain Experts’ Concepts Domain experts can also be test engineers, commissioning, configuration, packaging and deployment engineers, or service creators. Because they are usually not programmers, a modeling language for them needs to raise the level of abstraction beyond programming concepts. Languages that are based on domain experts’ concepts are relatively easy to define because for an expert to exist, the domain must already have established semantics. You can derive many of the modeling concepts directly from the domain model. The same holds true for some of the constraints.

Figure 10.4 shows a language based on domain experts’ concepts (see Chapter 6 for details). For this particular language, the modeling concepts are related to financial and insurance products. Concepts like “Risk,” “Bonus,” and “Damage” capture the relevant facts about insurances. Using this language an insurance expert, and thus a nonprogrammer, draws models to define different insurance products. Generators take care of transforming these designs into code for a web application for analyzing and comparing insurance products. In this way, the expert programmer can build the mapping from the language to the code once, and neither he nor the insurance experts need to know the intricacies of the others’ area of expertise. The higher abstraction in models using domain experts’ concepts also means that the generated output can be easily changed to some other implementation language.

Generated Output The fifth and last category of concept sources is the generation target of the language: the concepts and structures we see in the code to be generated are mapped directly into the modeling language. One of the most typical cases here is defining a metamodel based on the schema of the XML to be generated: Each tag type refers to a concept in a modeling language. While these languages are easy to build, their ability to increase productivity and quality is questionable. There is a danger of creating languages like class diagrams: presenting a class as a rectangle that maps one to one to a line in a file. This kind of language may still be valuable when the generated output is already in a domain-specific language, like a particular XML format. The XML schema will provide you with a wealth of information for identifying the modeling concepts and constraints. To follow the XML metaphor, designs can be considered valid and well-formed right at the modeling stage.

Graphical models can also help overcome many of the limitations of XML.

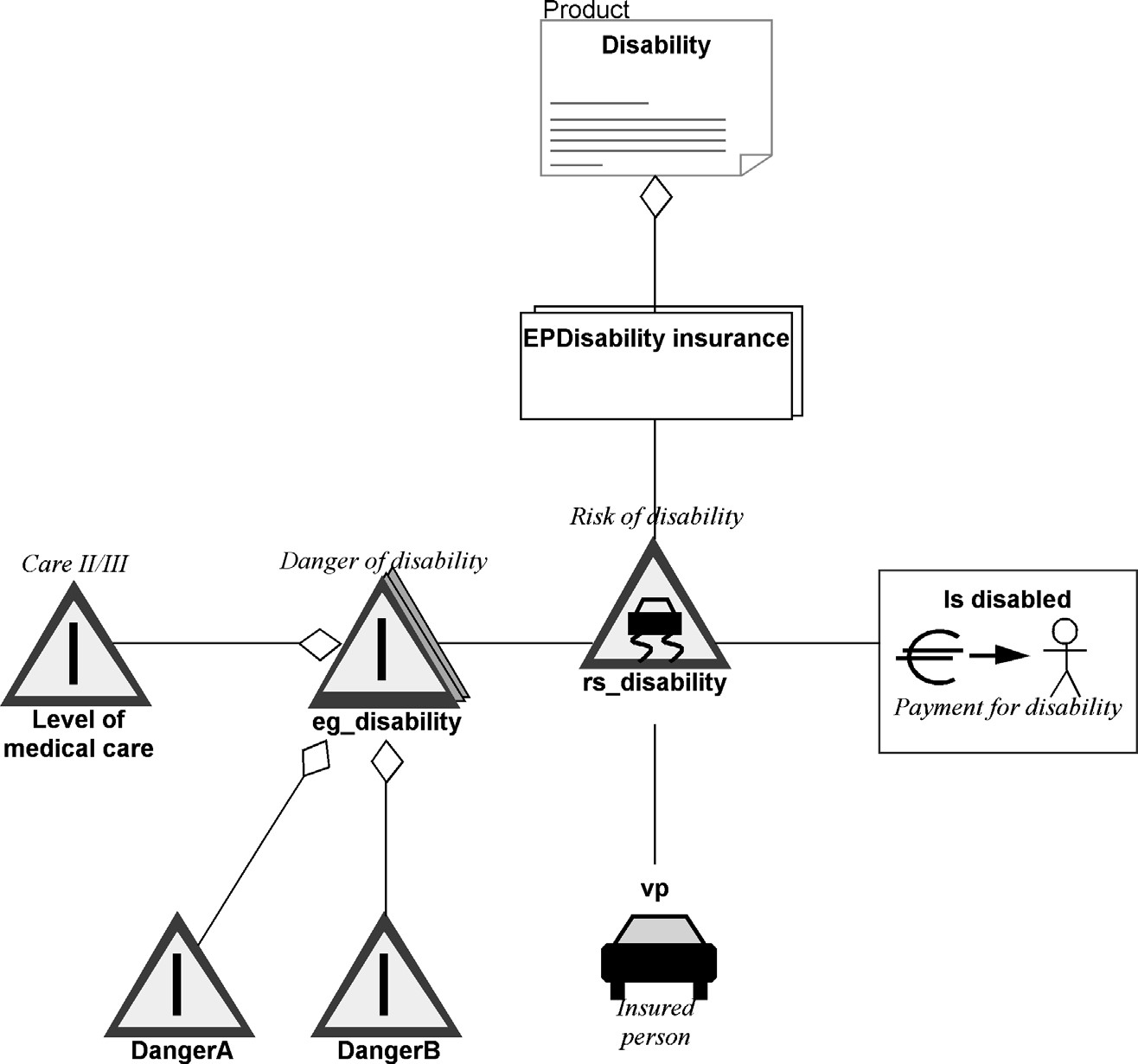


FIGURE 10.4 Modeling financial and insurance products for a J2EE web application with a DSM language based on domain expert concepts

An example of this kind of DSM language is the Call Processing Language (CPL), which is used to describe and control Internet telephony services (see Fig. 10.5 and Chapter 5 for a detailed description). The modeling concepts include “proxy,” “location,” and “signaling actions,” essential for specifying IP telephony servers. These same concepts are already defined as elements in the XML schema, and the property values of the modeling concepts are attributes of the XML elements. Having generators produce the configuration in XML gives significant and obvious productivity and quality improvements. With the modeling language, it is far more difficult to design services that have errors: something that is all too easy in handwritten CPL/XML.

10.2.3 Choosing Computational Models

Domain concepts do not exist independently: they relate to each other. These connections can be illustrated in the models too with supporting modeling concepts.

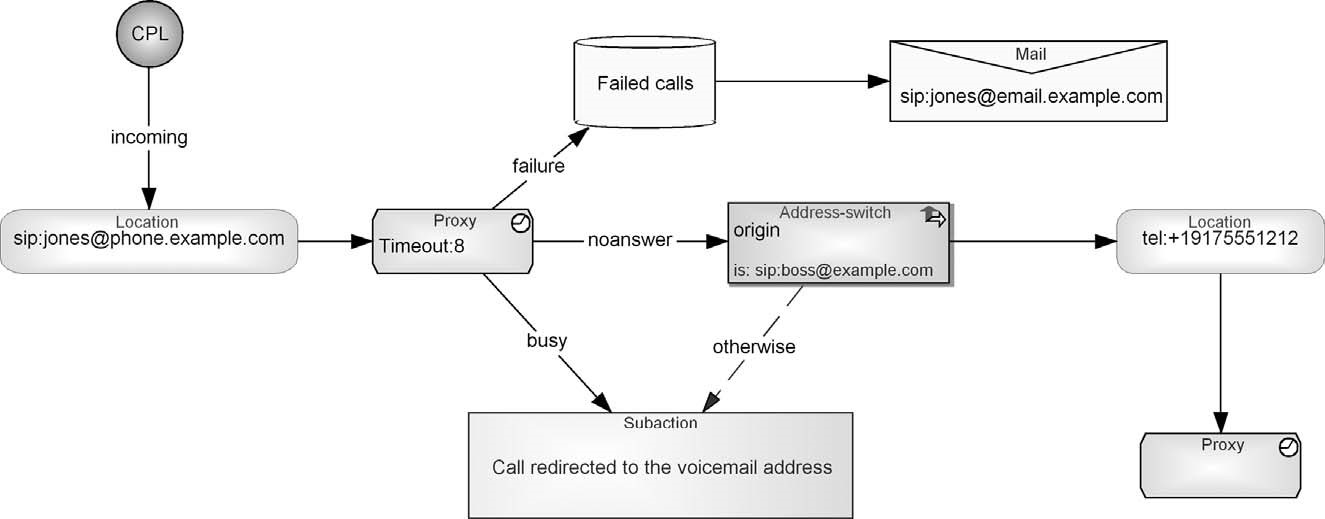


FIGURE 10.5 Modeling call processing in IP telephony, a DSM language based on its XML generation target

The way domain concepts should be connected leads to finding a suitable model of computation (MOC, see Section 4.2.2). Usually, the main approach can be identified relatively easily: should we focus on specifying static structures, behavior, or both. When inspecting details of the domain there are, however, big differences, and it is the details that matter when using models for code generation.

The insurance case described in Chapter 6 is the only example in this book of modeling just static structures. The rest of the examples also cover behavior, and some address only behavior. The watch example in Chapter 9 is based on using two different languages and models of computation. First, the physical structures of individual watch products are specified by defining their elements: displays, icons, buttons, and time units. The main part of the watch products is, however, the functionality of different applications, like how time, alarm, or stopwatch applications work. For this purpose, a state machine is used as a foundation for the second language. It is then extended with domain-specific concepts for specifying state-based and event-driven watch products. Both languages are further integrated by using partly the same concepts so that model data specified in one language can be shared with another. For some domains the static parts, like data elements, can have such minor roles that creating a separate language for just a few aspects is unnecessary. It is best to extend the behavioral language to allow specifying static structures. This was the case with the mobile phone application (Chapter 8).

You should not choose the MOC solely based on the domain. The expected generated code also influences the modeling language choice. For example, if most behavior is provided by the underlying framework, we can produce just structure and interface data. This suggests the use of languages addressing static structures and is often the easiest way to achieve model-based code generation. Although we can generate functional and behavioral code from models describing static structures (like schema creation for a database), the possibilities of specifying behavior, logic, dynamics, and interaction are easier with languages addressing behavior, such as various flow diagrams, state models, and interaction diagrams.

Library of MOCs The easiest way to start defining a modeling language is to base it on the computational model of an existing language—which already has some proven structures. Tools may help here by offering a library of basic metamodels to start with. You can then copy ideas, or sometimes even concrete elements of the metamodel, into your language and extend them with domain-specific concepts. Such concepts can be new modeling objects, their relationships, properties, bindings, constraints, model hierarchies, reuse rules, and so on. You have here numerous possibilities. Consider, for example, a relationship between model elements. It can be specified by its direction, multiplicity, n-ary, parallelism, or cyclic structures. Each may have further specification of details, like the maximum and minimum values for multiplicity; n-ary relationships can be further specified by the number of different, possibly optional, roles they have; cyclic relationships can further be defined to differentiate between direct and indirect cyclic structure; and so on. Communication flows can further be seen to be synchronous or asynchronous and have different policies, such as balking, timeout, or blocking. It is the numerous small extensions to the basic models of computation that make a modeling language domain-specific.

Multiple Languages and MOCs A single language and MOC may not be enough to carry out specification work. The domain can have multiple aspects each requiring separate views, there can be different roles among modelers or different levels of detail on which to focus, or putting everything into one language is simply not possible. We have not found strict rules on when to add new languages rather than continue adding new concepts to current languages. For example, in one telecom case, language developers had defined just two languages for a relatively large domain: one for static UI and another for the rest. The second language showed multiple aspects, such as data access, process, real time, and events, all in the same model.

When deciding on the number and type of languages, it is best to test the ideas early by making test models: instantiating the metamodel. You should note that new languages also mean that models need to be kept correctly integrated. Keeping everything in a few languages makes modeling work and consistency checking simpler when compared to spreading the specifications over multiple different kinds of languages or views. UML is an extreme case: with 13 separate diagrammatic languages, each having partly different modeling concepts, it becomes difficult, if not impossible, to keep specifications in synch. In DSM, the integration of the languages can be built in via the metamodel. For the telecom company, the decision to divide the domain into two isolated the UI changes from the rest. Even so, models made with the different languages had some integration and linkages. We will discuss language integration in more detail in Section 10.5.

Starting Language Definition from Scratch or by Modifying Existing Languages Although you may use a basic computational model as a starting point, it does not mean that you need to define languages by modifying already existing metamodels. Quite often defining the language from scratch—and naturally keeping in mind the basic models of computation—is simpler and faster to do. You don’t need to first learn concepts and semantics defined elsewhere, and you end up with a simpler language definition. Also, you know the language better as you defined it. This becomes important later when modifying the language, and also because you don’t need to consider changes made by others.

For example, all the languages in Part III were defined from scratch rather than by using existing metamodels or their parts. The only exception here is the mobile phone case, in which the modeling language used first for the Python framework was used for generating C++ for another case. It was, however, natural to reuse the already defined metamodel since the domain was practically the same. Modifying an available language can still be a good approach if the domain addressed is closely related to an existing language and changes to this language are minor. The Meta Object Facility (MOF) case in Chapter 6 resembles this situation, but using all the UML class diagram concepts and then choosing which of them are relevant during individual modeling situations would be too costly when compared to creating the language from scratch. Also, much of the class diagram metamodel (template classes, associate classes, etc.) was unnecessary and would introduce extra complexity for modelers. Often the limited capabilities of tools for creating new metamodels and modifying existing languages also partly dictate the language selection and modification.

10.2.4 Defining Modeling Concepts

Having identified the abstraction for the specification work along with some suitable models of computation, you should start mapping domain concepts to modeling concepts. While doing so, you balance between having a dedicated modeling concept and leaving decisions to the modeler. So the question you often need to answer is: should this particular concept be an instance value or a type? A string entered by a modeler or recognized by the language?

Next let’s consider an example: how should icons in a digital wristwatch (Chapter 9) be handled in the language? Should it be a particular kind of instance value a modeler enters into the model or a concept of a modeling language? The choices could be as follows:

1. Apply a certain naming convention to identify that this model element is anicon. The modeling concept used to specify icons is thus more general since it can also be used to specify elements other than icons.
2. Use some built-in language extension mechanism. In UML, this could be doneby giving a stereotype <<icon>> to the model element.
3. Have a dedicated Icon concept in the language. This Icon concept can thenhave the properties and other constraints relevant just for icons. For example, if the possible set of icons is fixed, the Icon concept can have a list of legal values from which to choose. If there are different kinds of icons they could be specified as subtypes having their own characterizing properties and possibly inheriting common ones from the main Icon concept.

In DSM, a language identifies all the relevant concepts, whereas a general-purpose language leaves most, if not all, for the modeler to decide. The latter choice then makes model checking, reuse, integration with other models, and code generation difficult. Consider, for example, integrating two models. You may use string matching by naming models or model elements similarly or have special properties or other naming concepts to indicate the integration. The other alternative is to recognize integration already at the language level: the modeling concepts are the same or share the same details. Using two UML diagrams as an example: Class diagrams and state transition diagrams clearly share some information, such as operations in a class and events of a state transition. This integration can be left to the modeller, or the modeling languages (metamodels) can be integrated. In the latter case, the name of the operation in the class diagram is the same modeling concept as the event name in a state machine. Since the metamodel knows that these concepts are the same, it is possible to check them, and when the name needs to be changed in one place, its change can be propagated elsewhere. The metamodel knows how the languages are integrated.

To achieve the benefits of DSM, you should in general always seek the possibility of having language support. This gives first-hand support allowing errors to be found and even preventing them early, guiding during development work, supporting reuse, and so on. Modelers then don’t need to learn all the details, and it becomes easier to define a code generator as you can be more sure that the input for code generators, models, is correct. If you leave domain extensions to modelers, everyone needs to know them—yet they still do them differently! There are also multiple ways to specify these extensions, such as naming conventions, annotating and commenting models, or using additional languages such as constraint languages or action languages. These choices depend partly on the tool chosen for language specification and language use (see Chapter 14 for tools).

Mapping Domain Concepts to Different Modeling Concepts Domain concepts can be mapped into different kinds of concepts in modeling languages. The usual starting point is that each domain concept maps one-to-one to a modeling concept. The main concepts often act as objects existing more or less independently from others. When adding more details, the focus generally moves from objects to other kinds of language concepts such as their properties, connections in terms of relationships, roles, or ports the objects may have in different connections, submodels, and even links to other model elements expressed in other languages.

The selected language type (e.g., state machine) and its MOC help to determine what kinds of additions are possible. You enrich the chosen base languages with domain-specific concepts and rules. To illustrate the use of alternative modeling concepts, let’s inspect another example from the watch case: Consider supporting a new button pressing policy in the language. Instead of pressing a button only once to start an operation in a watch, we have a new policy that is based on keeping the button pressed for a longer time, like 3 seconds. How should that be expressed in the language? It could have any of the following:

1. Its own modeling object: A long button press object in addition to the existingshort button press.
2. A property of the current object: The current button concept in the languagecan be specified as either a short press or a long press. If the short press is the most typical case, the property could have the corresponding default value. Alternatively, if the value is left unspecified, a code generator could produce the default value.
3. A role (or relationship): Different ways to press a button can be specified as anew role or a relationship, keeping the button concept in the language unchanged.
4. A property of the current role: The pressing policy could be specified byadding a property for the current role that represents a button press.

Other kinds of extensions could be considered, like having the button pressing policy specified in the family diagram in which the button is first introduced. This concept, although hardly relevant for our sample case, would keep the behavioral designs untouched and move the decision to each individual watch product.

View of Variation Creation of a DSM solution should also be inspected from the variability point of view: how differences among applications can be specified. This is not always a topic for language definition since the variation can be handled elsewhere, typically in a generator. This makes management of variation simple for developers, as it totally hides it from models. A developer just chooses among different generators, which actually access exactly the same set of models to produce the code. For example, the mobile phone case uses the same modeling language but different generators to produce either Python or C++ code. Similarly, the digital wristwatch case illustrates the use of the same modeling language but with different generators, such as one for Mobile Information Device Profile (MIDP) Java and another for C. This offers the benefits of having a single source and multiple targets: the cost of creating applications for other target environments is almost nonexistent. Also, bug corrections and other changes need to be made in one place only and newer versions can be generated for all the targets.

Usually in such cases, the underlying target environment is different, as are the supporting components and framework code. In the case of the Symbian smartphone, there are two options: when creating Python applications the generator uses the Python for S60 library and produces corresponding framework code such as dispatcher and stack handling code; to produce C++ code, native for Symbian, the generator instead uses the UI platform of S60. The framework code can be produced by the generator or it can be selected from the library during generation. The same rules apply here that applied to selecting components: Each component that is optional or whose use is based on variability can be represented with variation data. For example, in the watch case there are two different Canvas components for Java implementation: one for running the application in a browser and another for MIDP devices. These components are then characterized with an additional property for specifying the platform, i.e., “applet and awt” or “midlet.” Depending on the target, the generator then reads the platform properties given in models and includes the right components along with the code generated from the models. The same principle is also normally followed in cases where there is a need for different generators for different purposes, like one for early prototyping, one for producing code with model debugging information, and one for generating production code.

Supporting Variation in Languages Most domain engineering approaches (e.g., Kyo et al., 1990; Arango, 1994; White, 1996; Weiss and Lai, 1999) emphasize language as an important mechanism to handle variation. This means that variation is represented right in the metamodel. In the simplest case, a variation point is defined as a property of a modeling concept. The possible parameters of variation can be further defined as predefined values from which a modeler chooses the right one. In the watch case, for instance, each display element can be specified by the icons it may use. These are defined in the metamodel as a property type having as predefined values the icons that can be used. A modeler then just picks one or more of them depending on his need. The parameter can naturally be more complex than just a single value, like an object having additional properties, a graph, or even a set of graphs. In Fig. 10.6, the display function showing clockTime illustrates such a parameter selection. The design on the left shows time in minutes, seconds, and milliseconds, whereas the application on the right shows time in hours, minutes, and seconds. This choice of the central time unit is made from the list of possible predefined values.

Rather than having separate properties for illustrating variation, the main language concepts can be applied to describe variation. Every model element added to the specification can originate from the variation. If we again use the watch case, since it is the best case of a product line among the examples in Part III, placing an object type like “Alarm” means that the application has alarm functionality. Further, variation can be represented with model connections. By connecting the alarm element, we can describe how the alarm is set and what happens when an alarm rings. In the same way, the transition relationships illustrate the variation in execution order within each application. The same transition relationship is also used to specify execution order among the applications for a particular product. For example, Fig. 10.6 illustrates two different watch models: one with three applications and another with five. The variation is specified here by having different application states as modeling objects and by connecting them in different orders with relationships.

In the above case, each application state is further described in additional submodels: each state has one subdiagram describing how each application, such as time, stopwatch, and timer, works. This illustrates the option of having complete diagrams as sources of variation. If the same stopwatch application cannot be specified for two cases, the language then allows making different subdiagrams for each case. This option means that two different designs are created. The obvious drawback is that the application specifications most likely have some similarities but are still created separately. This not only means that there is duplicate work in the beginning but also that possible bug corrections and extensions that are beneficial to both applications need to be made multiple times: here twice, to cover both models.

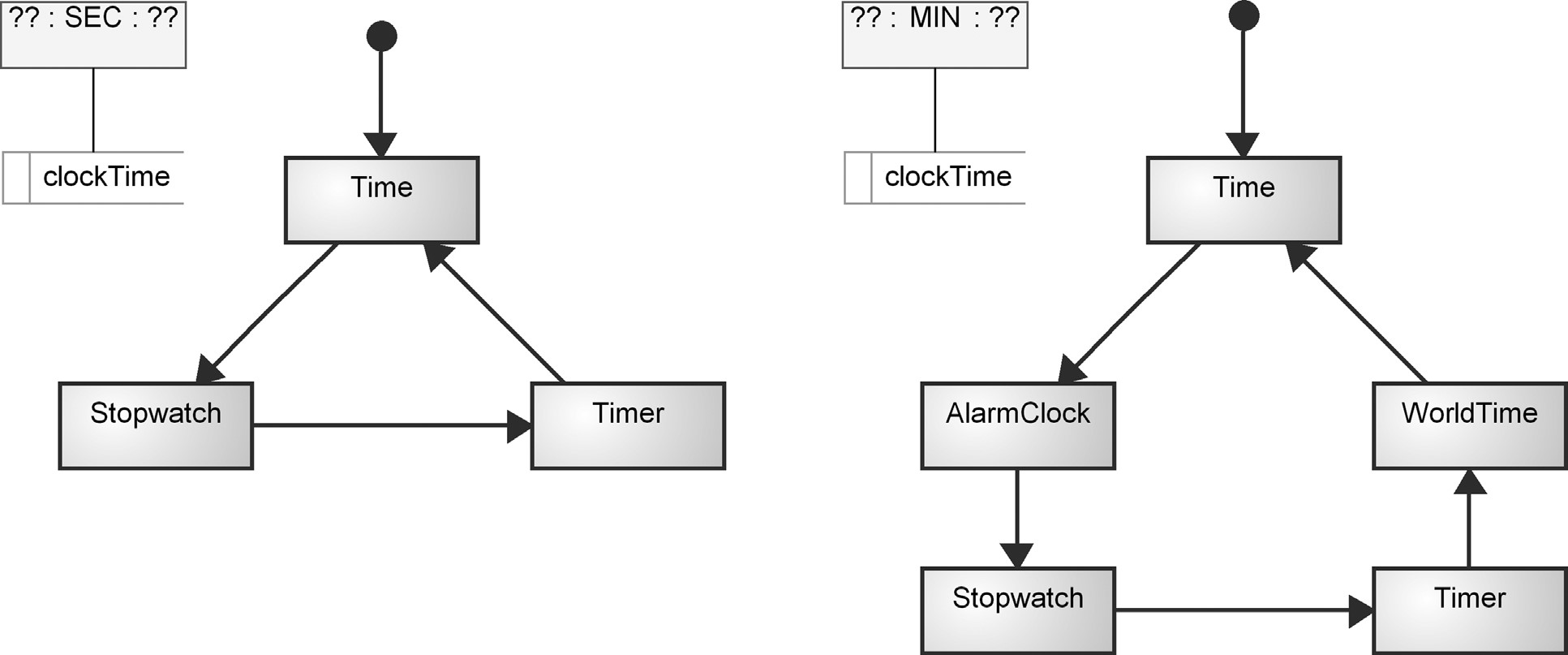


FIGURE10.6

Twodifferentmodelsdescribingtwovariantsbasedonincluding

applicationsandtheirexecutionorder

244

Capturing variability does not need to be limited to specifying just added functionality. Modeling concepts can also be used to remove, or more accurately to ignore, certain model elements during code generation. Following the same language structure as above, one way is to have a property for those elements that can be optionally ignored. The modeler can then just change the property values for those elements that are not included in the generated code. If the excluded elements are connected with other model elements, then they can be excluded too. An alternative mechanism to ignore some model elements is to connect them to a model element that is used just for excluding parts of the design based on the required variability. Such exclusion elements may have their own properties that the generator reads while selecting the exclusion elements and ignoring the related model elements during code generation. Often, though, having multiple relationships just for exclusion may make the model complex since it now has both application data and exclusion data. This becomes even more complex if there are multiple variability objects, one for each variation point. Then, it often becomes better to create a configuration diagram just for describing the variation. This variation diagram then usually works also as a starting point for code generation: it is related to other models and the generator already knows the desired variation when accessing the actual specifications.

Modeling Concepts Should Also Support Editing Language definition should not focus only on the final static model but also cover the use of the modeling language. This means thinking about the model editing process, how models can be kept consistent, and how reuse of model elements can be supported. In our example of button pressing, having different button objects for a long press and a short press would make some model editing actions unnecessarily complex. Simple refactoring actions, such as changing the name of the button, could mean editing the name twice: for both button types when two different policies are used. It would also become more difficult to check the consistency of the models: what if only the short press button name is changed from “Mode” to “Set”? How could we inform the modeler that the “Mode” long press needs to be changed too? Defining the button with information about its usage would limit reuse possibilities and increase modeling work: we could not reuse the button specification so often anymore as it would always come with its usage information. While modeling, we would then need to specify the button multiple times if it is used differently in the same application. A closer mapping to the behavior in the problem domain would lead to reuse and minimize modeling work. Having the button pressing policy specified in a role allows reusing existing button definitions and specifying its usage information separately only when relevant: The same button element could then be used multiple times in models and have different usage situations.

Language Definition Guidelines Typically, the best way to start language definition is to extend the selected MOC with basic domain concepts: first, add the most essential and most used language elements. Then, continue by identifying their connections along with the various rules and constraints. As most of us will be defining our first language for the domain, it is important to test the modeling concepts early. This means trying out the language early on. Having a working language, albeit limited in expressive power, and gradually testing and extending it makes language definition agile.

You don’t need to define all the rules in the beginning as the modeling concepts could change. The same applies for notation and generators. Implementing them too early, when the language is not yet stable, can waste time and effort. Some guidelines for language definition are as follows:

. Follow established naming conventions: While defining the concepts, it is usually best to use exactly the same names and naming policies for the language concepts as are already used. Sometimes during code generation you may need to follow a particular naming policy (uppercase, lowercase, etc.) to map to the selected implementation target. Although it is possible to make the generator do the translations, defining a DSM solution becomes easier if the naming policy is supported by the language. This applies not only to checking values entered in the models but also to naming modeling concepts. For example, in the IP telephony service example (Chapter 5), the naming of modeling concepts is taken directly from the domain and expected output. The naming applied in language types can then be straightforwardly used by the generator to produce names for XML. Having multiple implementation targets, however, often means the generator translates model data to the given target language.

. Keep the language simple and minimal: One of the most typical errors is trying to cope with all possible imaginary scenarios or make the language theoretically “complete.” This makes things unnecessarily complex and increases the burden of defining a higher abstraction. It is best to stick with the identified needs and support them first. You can always extend the languages later if needed. If we stay with the watch example, the language for logical behavior now supports basic arithmetic operations for time, but for the sake of covering a larger set of operations, we could also add, or example, multiplication and division. But since these operations are not currently needed, supporting them would just take extra effort with no immediate or long term benefit.

. Try to minimize the modeling work: Don’t ask modelers to fill properties you already know or can infer: they can be produced by the generator or provided by the domain framework. Follow the principle of using convention over configuration. For example, in the mobile phone case (Section 8.3.2), the need to specify cancel navigation is made unnecessary in over 90% of the cases, as a default target for canceling can be produced by the generator. Still the modeling language has the concept for specifying the cancel navigation if the default path is not appropriate.

. Have a precise definition for each modeling concept: When defining modeling concepts you need to precisely define what each concept means. This is unlike in most general-purpose languages, in which semantics are vague and left to every individual model creator and reader to define. Code generation expects that you know what you are automating. You should have examples illustrating the alternative cases, the concepts, and how they behave.

FORMALIZING LANGUAGES WITH METAMODELING

. Consider language extension possibilities: If the domain is new or it is unclear whether the defined language provides the needed modeling capabilities, you may add special extension concepts to the language. For example, add to the language a modeling object that can be connected freely with other modeling concepts and that has just one description property. The code generator can then skip these model elements, yet we can inspect what additional modeling needs there may be. Alternatively, we may limit the extension to only a few places, in the modeling concepts that it makes sense to extend. A special case of this is having a “code object” in the language, including plain code or referring to external sources having the code. For example, in the mobile case, the form validation function is made open in this way: a developer can enter any Python script in the validation function and that code is then included in the generated code. Clearly, this must be used sparingly to maintain a high level of abstraction.

Every domain concept (and variation point) does not need to be in the language. Some relevant domain concepts can be “composed” by combining existing concepts. For example, the mobile phone case applies the state machine as MOC where navigation shows that the state changes in the application but the language still lacks traditional transition characteristics like event triggering the transition, a condition that needs to be met and the action performed during transition. Events are already limited to a few possibilities, like pressing predefined softkey buttons, and a condition is left to be specified only in a few cases where a selected value is used as a condition for choosing a specific navigation path.

10.3 FORMALIZING LANGUAGES WITH METAMODELING

As soon as you have identified the relevant parts of the modeling language, you should formalize it. This is best done by defining the metamodel. Formalization into a metamodel is needed because otherwise the language does not guide the modeling work and code generation would not be possible (see Section 4.2.4 for details). We need to emphasize here the difference from the definition of UML, which is also described as a metamodel. The metamodel of UML is a collection of separate, loosely connected class diagrams that are not instantiated and tested during language development. Partly for this reason tool vendors implement support for the languages of UML differently. They may look the same, but inspection of the details shows the difference. You can avoid this by using a metamodeling tool that can also execute the metamodel. The metamodel also serves as a basis for integrating and sharing the models with other tools in the chain.

10.3.1 Metamodeling Process

Metamodels are often best specified first with just pen and paper in some format. This usually means drawing a data model showing the modeling concepts and their connections. Often companies use an entity-relationship diagram or a class diagram in this initial specification. If a class diagram is used, do not specify details, such as operations of classes and relationships other than associations and inheritance. When you know the structure to some extent, it is time to define the metamodel.

Metamodeling simply means modeling your language: mapping your domain concepts to various language elements such as objects, their properties, and their connections, specified as relationships and the roles that objects play in them. This process is supported by the metamodeling language which, needless to say, should itself be a domain-specific language for specifying modeling languages. The metamodeling language depends on the DSM tool you use, but at a minimum it should allow you to define the concepts of your language, their properties, legal connections between elements of your language, model hierarchy structures, and correctness rules. In all but the smallest cases, support for reuse and different model integration approaches is also essential.

Language for Metamodeling A good metamodeling language guides you during language definition: it allows you to focus on defining modeling concepts and hides the implementation details (how to run it in editors). During metamodeling, you specify some of the language concepts directly and others by combining some domain concepts. In making a decision about which concepts to include, it helps to use your language for modeling and generate artifacts from it. Here, tools can help you, as ideally they should allow you to focus on language definition only and providevarious modeling editors for your language instantly and automatically. This makes language creation agile: you can easily test and learn what the language looks like in practice, how it allows you to make and reuse models, and so on. This minimizes the risks of making a bad language, or a good language but for the wrong task, and helps in finding good mappings for code generation. This kind of prototyping is best when language design is something new for you or the domain is not yet completely defined.

Metamodels for the Language Definer, Models for Others The possibility of trying out your language specification immediately after you have defined some of the concepts significantly changes the role of metamodels. First, they are formal: you can run them. Next, they formalize the domain knowledge in such a way that other developers and stakeholders can understand it too. They may not be able to understand the metamodel, but they will understand models created using terms they are familiar with. Concrete example models allow to show the idea to other developers early and make it easier for them to understand the ideas and contribute to the language. This will greatly support language use later, in the introduction phase.

10.3.2 Sample Metamodeling Task

Let’s next inspect the metamodeling process with an example. Consider the case mentioned earlier of supporting different button pressing policies in a modeling language specifying watch applications. The task of the language designer is to choose the most suitable structure for the language and define it into a metamodel.

FORMALIZING LANGUAGES WITH METAMODELING

The options presented in Section 10.2.4 were creating a new modeling object (long button press), adding a property to a current button to choose the policy (short or long press), adding a new role to the language (long press event), or adding a property to the current Event role for different pressing policies. Since the first two options do not support reuse, and the same button would be used in every case with the same policy, it is best to modify the role. This allows us to use the same button in different ways: sometimes the button is pressed briefly, and sometimes for a longer period. Rather than making the language larger with its own dedicated role type for long pressing, we decided to add a property to the current Event role.

Figure 10.7 illustrates this change. This metamodel defines that there can be a transition from a state to another state or a stop state. A transition can be activated by an event caused by a button, and it can initiate an action. You may compare this to the metamodel illustrated in Figure 9.9. The only difference is that the Event role now has a Boolean property called “Long press?” Its default value is false since a short press is the more common case.

Once the metamodel is updated, we can next try out the language. Depending on the tool (see the different alternatives in Chapter 14), you may need to generate the metamodel for another tool or just use it immediately for modeling. Again, depending on your tool you may need to define a new model or the tool can update the existing models automatically for the changed language. Figure 10.8 describes a sample model, in which the long press policy is used for the Down button. An ellipse and related text are used in the notation to visually indicate the difference in the model. Using both is perhaps overkill, but something is necessary to help read the model.

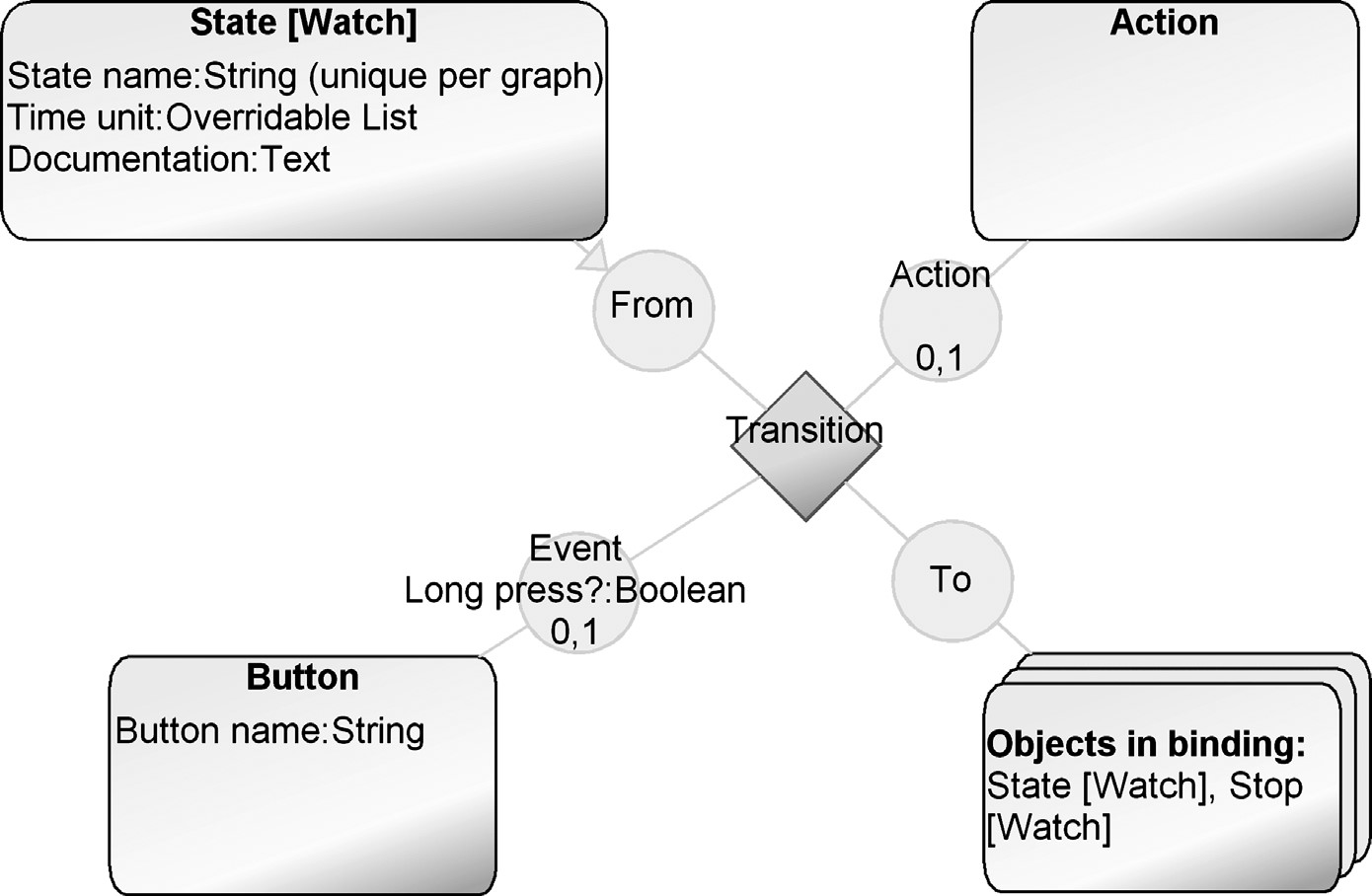


FIGURE 10.7 Metamodel having support for alternative button pressing policies

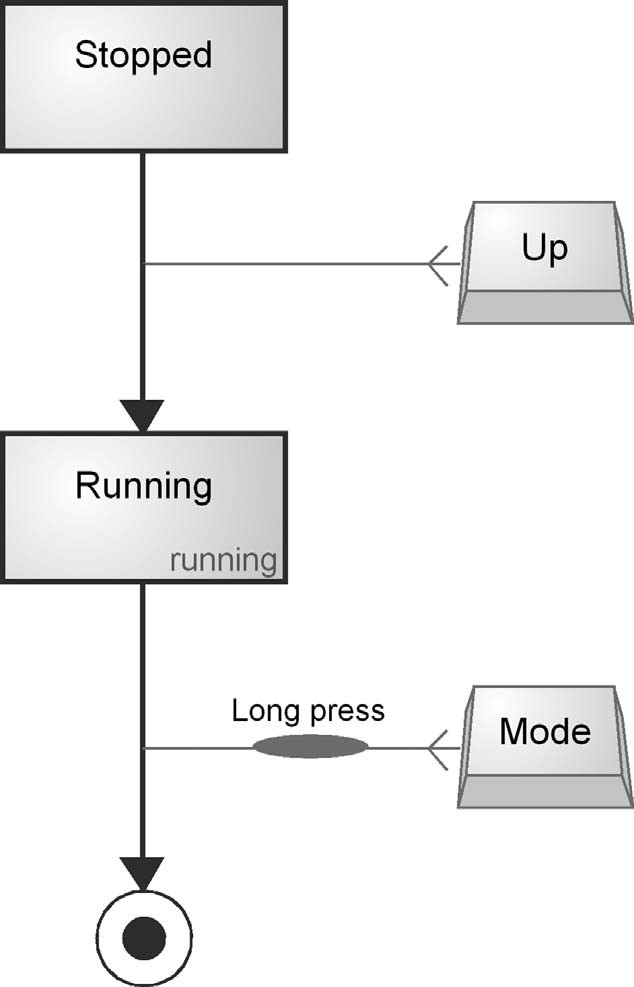


FIGURE 10.8 Sample model that is an instance of the metamodel specified in Fig. 10.7

10.4 DEFINING LANGUAGE RULES

Along with modeling concepts, we also normally detect various domain rules, constraints, and consistency needs which a language should follow. These rules obviously need to be defined too. Having rules in the language provides many of the benefits of the DSM approach: they make a language domain-specific. Typical benefits are as follows:

. Prevent errors early: illegal or unwanted models simply can’t be made.

. Guide toward preferable design patterns.

. Check completeness by informing about missing parts. While not necessarily relevant for the modeler, the code generator expects a “complete” specification. It is worth noting that by “complete” we mean that models can be used as input for full code generation.

. Minimize modeling work by conventions and default values.

. Keep specifications consistent: if one element is changed in a model, the change is reflected elsewhere to either update models or report about the inconsistency.

In our experience, the rules are best defined after having decided on the main modeling concepts: then you start to see the patterns and rule types. It is not rare to notice that a certain rule, such as occurrence, naming, or cardinality in a relationship, is

### DEFINING LANGUAGE RULES

shared among many modeling concepts. We can divide the rules into those that originate fromthe domainand those thatdealwithhow the modeling languageoperates.

10.4.1 Domain Rules

Most of the concepts in a modeling language come with rules. These rules should naturally be recognized by the language too. The types of rules you can specify depend on the DSM tool and metamodeling language applied. We discuss the rules here independently of any tool, but the types of rules that occur most often in modeling languages deal with the following:

. Naming conventions, e.g., a value must start with a capital letter or must not include certain characters

. Uniqueness, e.g., there can’t be another element with the same property value

. Mandatory properties forcing an element to have a value

. Default values

. Occurrence: a concept can only have a certain number of instances in a model . Binding rules stating which kinds of elements can be connected together

. Connectivity rules stating how many times an object may have a certain kind of connection

. Reuse rules stating that a modeler can choose a certain value or refer to another model element

. N-ary relationship rules stating how many objects a single relationship can connect

. Integrating models, such as sharing the same value with another element, possibly in another model, and possibly made with another language . Model structuring rules, such as hierarchies or references to libraries

You may identify these rules from the same sources you identified the domain concepts. However, detailed language rules often can’t be detected from existing material, and the fastest way is to check them directly with the domain experts. You may also look at the kinds of rules that the selected model of computations uses.

10.4.2 Modeling Rules

Strict enforcement of domain rules can sometimes conflict with the needs of modeling. While the domain rules as such are obviously correct, they may not always work well as rules of a language. Usually, this means that the usability of a language would suffer if domain rules were used as-is. A classic example is forcing the removal of an element, like a relationship, before a new one can be created. While correct by static domain analysis, it can often be better to temporarily permit the inconsistency, allowing modelers to see the old information while entering its replacement. Past information is often useful when creating new model elements.

It may not always be possible to check domain rules in a language. The number of model elements and models, possibly made by other developers at the same time, could be so large that checking rules after each model manipulation would require too much time, hindering modeling work. Checking some rules, like uniqueness among all model elements, can be time consuming or often impossible. It is, therefore, sometimes better to allow illegal models for a while. For example, checking minimum cardinalities at modeling time does not normally make sense since it would prevent modeling work. Adding a start state to a state transition diagram would create an illegal model since a start state needs to be connected to some other state, making it the initial state. In contrast, maximum connectivity, checking that there is no more than one transition from a start state, is good to put in the metamodel: it ensures correctness without interfering with the normal flow of modeling.

Checking and Informing About the Rules Rules can either be strictly enforced in the metamodel or shown as warnings by a separate model check. Placing rules in the metamodel is usually the better choice since it keeps specifications legal at all times. The worst option is to check the rules just before code generation since then there is no support during the actual modeling.

Depending on the tool, the results of rule checking can be visualized in different ways. Ideally, the best way is to give immediate feedback after modeling creation tasks. This necessitates that a model, and with some rules all models, need to be checked after each model manipulation task. In practice, this only makes sense for certain model manipulation tasks, such as creating a relationship or changing a property value. If immediate feedback is not possible, then model checking can be done as a separate process, typically when the modeler decides to do so or when a code generator is executed. Here, modelers can be informed in different ways about the results. The most natural option is showing the warning close to the model element or highlighting the elements the rule is related to. For example, an element having an error or not yet completely specified could be selected or reported. Each modeling action can also inform about the possible illegal action or even inform the modeler about other options of language use. The last option is to have a separate checking report showing the results in some textual form, like an error dialog. Tools may also help in the visualization, such as by allowing checking results to be traced back to the model or model element having an error.

10.4.3 Rule Definition Process

In general, the rules of a language should work like file security: nothing is permitted unless it is specified. The advantage of this approach is obvious: language definition becomes easier to handle, as new rules can be set and tested incrementally. This is the exact opposite of using profiles in UML: things are already allowed and then we start adding extra rules to override those already defined in the standard metamodel. This not only complicates the definition but also makes the rules cumbersome and numerous. Try it yourself: implement the languages described in Part III using plain

### INTEGRATING MULTIPLE LANGUAGES

UML as a starting point and adding rules with profiles. You will quickly find why metamodels are simpler and easier to define.

Remember that rules can evolve over time too. Modelers usually require more expressive power and the domain itself changes. For example, if modelers can’t specify some functionality, you may relax a rule and move its checking to the generator. Alternatively, if unwanted structures or specifications that lead to poor performance are identified, you may always add rules to the language or check the models before starting the code generation. A DSM tool should then allow updating the models made with the earlier language version along with the metamodel.

10.5 INTEGRATING MULTIPLE LANGUAGES

A good design language is not isolated from other views modeled with other languages but is integrated. This is nothing new: the software we develop today for manipulating data acts in a similar fashion: when a loan application in a bank uses account numbers, these are not saved separately for this application but are shared with others, such as the ones managing the accounts, calculating interest ratios, and so on. Specification data in models is no different: its integration needs to be specified similarly. You may integrate models specifying different views of the application, or integrate models to existing code libraries, to other models describing variability, or to models specifying nonfunctional requirements. As a language developer, you can build this support right into the language. You may also consider doing integration via transformations, but that seldom works, as explained in more detail next.

10.5.1 Integration Approaches

The best way to integrate languages depends on the number of views, how model data needs to be available to developers, and whether there is a need to keep developers in separate modeling spaces or modeling is based on reusing others’ work. Figure 10.9 illustrates the most typical approaches.

The first approach is to keep the languages separate and integrate them at generation time. This approach makes things simple for the language developer but more difficult for modelers and generator developers. This is reasonable if you need to

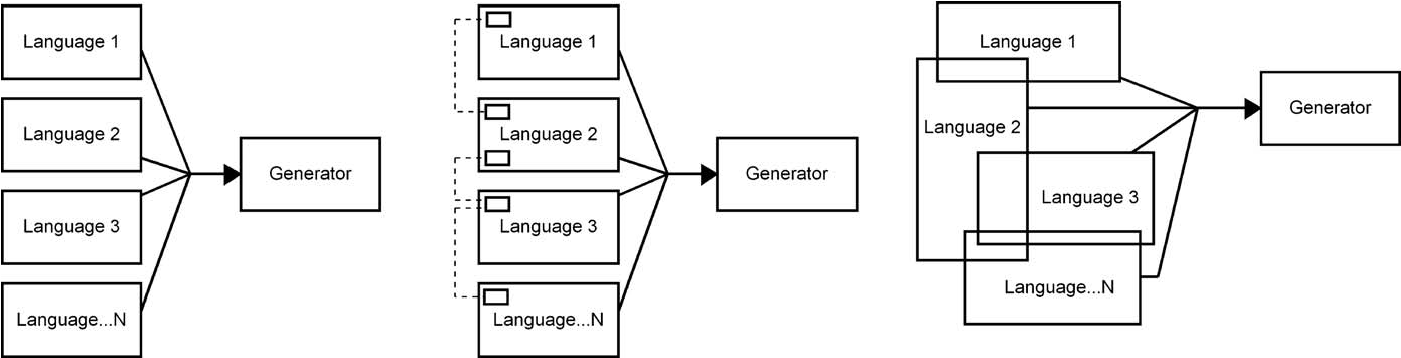


FIGURE 10.9 Language integration approaches

keep the modelers separate or they don’t need to interact. For example, while using subcontractors you may decide to keep some models for internal use only and have another language for subcontractors. It can also be that the nature of modeling work differs among developers so that all groups don’t need access to others’ work. This approach naturally makes generator development more difficult as it can’t guarantee via language rules that models are correct. The generator then needs to run the checking as a separate process or, more likely to check the input internally without providing trace links back to models.

Whenever possible, it is better to support integration at modeling time—it enables integration before the generation. The second approach is to define concepts enabling model integration. Stahl and Vo¨lter (2006) suggest the use of a few concepts that are common between two or more languages. These are called gateway elements in the metamodel. These few special modeling concepts are then used to link separate languages (and models) together during generation.

In reality, the domain concepts are integrated and related, whereas the integration concepts are just additional reference elements that need to be handled separately. Therefore, it is usually much better to define that languages share and refer to the same modeling concepts. This means integration based on the metamodel. You may share the same modeling concepts completely or just parts of them, like some of their properties. You may also integrate languages based on model decomposition, have another language for specifying details of one particular model element, or define other mappings between modeling concepts. While this is more difficult for the language developer, it makes the life of modelers much easier, and generator developers can better rely on the input for generators being correct: it is checked in the models. For modelers, the integrated metamodel gives a fundamental benefit: They can see other models, and models update automatically based on changes made elsewhere. Data in models can then be edited in an integrated way and there is no need for copying and pasting elements between different models or maintaining references among models. If the DSM tool supports multiple users, the change can be simultaneous for all developers. Also, model partitioning and versioning can be supported by allowing work with models consisting of multiple integrated specifications, rather than versioning small specifications separately and externaly keeping track of their dependencies.

10.5.2 Why Integrating Models with Model-to-Model Transformation is a Bad Thing

Experiences with model-to-model transformation, whatever tool is involved and whatever export/transform/import method is used, show this is normally a Bad Thing. You should only consider it when you don’t need to change the automatically produced model. Let us explain why.

Normally, the idea is that each piece of data in one model gets transformed to more than one piece of data in the second model (let’s say two pieces). This is fine if you never (or rarely) look at the second model and never (or very rarely) edit it. But if you edit it, you are now working with two pieces of data, but clearly they are not totally

### INTEGRATING MULTIPLE LANGUAGES

independent, since they could be produced from one piece. The idea of DSM is to come up with a minimal sufficient representation of systems, and this is the main reason for its 5–10 times productivity increases: code- or UML-based ways of building systems involve lots of duplication—the same information in several places.

People asking for model-to-model transformation say next that “we want to be able to change the first model still, and have those changes reflected automatically in the generated model.” That means you are working with 1 + 2 pieces of information, and also that you need to come up with some way to propagate the changes down to the second model “correctly.” “Correctly” here means without destroying information manually added there, updating the automatically generated parts, creating newly generated parts, and updating manually added parts to reflect changes in the first model. This last case happens if the manually added part refers to the name of an element originally from the first model and that element’s name has now been changed.

Next people say, “and we want to change the generated model and have the first model update automatically.” This is even harder. The only cases when it can happen are where you don’t really have a first model and a second model but rather two different representations at the same level. Even then, it can only apply to the intersection of the sets of information recorded in the two different models. For example, in UML tools you may have one class in a model mapping to one class in the code. With sufficiently simple changes, a bit of luck, and a following wind, the best UML tools today are capable of maintaining that simple mapping for the names of classes, attributes, and operations. The actual code isn’t kept in synch. Although tool vendors tend to claim that UML models are kept in synch, this generally means only some parts of class diagram elements. Synchronization is partial simply because the details of the code are not in those models that the tool can keep synchronized. The only way the synchronization can be made “better” is by moving the two languages closer: for example, by allowing UML operations to contain the method body as text, or the code to show things like “boundary class” as specially formatted comments. Each move toward better synchronization is thus a move away from having the first language on a higher level than the second language.

The DSM solution is to turn the question on its head and ask, “OK, you showed me the model, possibly based on a high-level modeling language, that doesn’t yet capture enough information to build full systems. And you showed me a second model and a transformation into it. Now tell me what extra information you want to put in the second models.” Note that here we’re asking for information, most likely on a (problem) domain level, not for the actual representation of that information. “Now let’s look at how we can extend or change the high-level modeling language to provide places to capture that information.” (Sometimes this step may require rethinking the modeling language, especially if it’s been based on a language that somebody had already made.) “And finally, let’s show how we now have all the information we need, and we can generate full code directly from the first model, using the information it originally captured and the new information we would otherwise have entered in the second model.” You should remember that there was only a certain amount of information to be added to the second lower-level model, regardless of its representation or duplication into several places there. Since we would earlier have been able to generate the initial second model from the information in the first highlevel model, add extra information, and then generate code, we can now clearly expect to be able to generate full code straight from the first model.

Summa summarum: if you can specify a transformation from a high-level model to a second, lower-level model and then have modelers add more detail and perform another transformation into code, you can look instead at the information that gets added by modelers, extend the high-level language used to specify the first model to capture it, and merge the transformations into one. It makes life easier for modelers since they have just one language and one model. There is no round-trip synchronization hassle or need to manage different models, their sharing, and versioning. It also makes life easier for the creator of the DSM solution: one language, a single one-way nonupdating transformation. Additionally, you normally find a way to record the information into significantly fewer data elements in the higher-level language.

10.5.3 Reuse with Models

Reuse of models and model elements can be recognized as first class citizens by the modeling language. This means that the metamodel knows when and where to find reusable model elements instead of creating them again. Reuse at the model level is naturally desirable as reuse can then happen at a higher level of abstraction rather than at the code level, and it can happen at modeling time. This means that reuse is built in to the metamodel.

The first approach to reuse is simple string matching: referring to a model or model element by typing its name in another model. This approach is possible in every language, but such reuse is not guided or enforced. Modelers need to enter particular values, usually following some naming conventions, and the generator makes the mapping and finds the reusable elements to be included in the generated output. The modeling language does not “know” when it would be better to reuse existing model elements instead of creating new ones, where to search for reusable elements, or whether reuse is based on a white-box approach showing internal details or a black box showing just the public interface.

A more sophisticated way is to share some of the same modeling elements between the models and modeling languages. Reuse of model elements in different parts of the same model is illustrated in the mobile phone case (Chapter 8). The return variables that store the values entered by the user are also used as input for other modeling concepts. For example, the SMS sending object has a variable message element that refers to any specified returnvariable. This allows the definition of SMS sending to use any variables entered earlier, and if the name of the returnvariable needs to be changed later its users, like SMS sending, don’t need to be updated manually. The metamodel takes care of that refactoring. Reuse is not limited to single values but also can involve whole models or their parts. For example, in CPL (Chapter 5) the subaction concept refers to other diagrams. This allows reusing any call processing service already defined. The watch example (Chapter 9) illustrated reuse of selected model elements

### NOTATION FOR THE LANGUAGE

rather than whole diagrams. The family model specifies the static elements of a watch model, such as buttons and icons. Later, in the behavioral model describing a logical watch, those buttons and icons could be directly reused. Changing the button in the static product component specification then automatically updates all the behavioral specifications that use the button.

Reuse always takes place in some context and direction and these can be specified in the DSM language. For example, the metamodel can make it impossible to create new model elements and enforce reusing instead. More flexible would be first guiding to reuse but allowing new model elements if suitable ones are not available. Finally, the language may be defined so that reuse is not mandatory and can be specified later once the reusable model or model elements become available.

10.6 NOTATION FOR THE LANGUAGE

Notation gives a visual representation for the models. Defining the notation makes sense only after the modeling concepts are already identified. Notation is therefore the wrong place to start defining a DSM language. Although model representation plays a minor role in code generation and other automated tasks, it is highly relevant for acceptance and usability. Especially in the beginning when a modeling language is new, there can well be more comments and feedback on how a language looks than on how it works. The creators of a DSM solution may feel a bit disappointed seeing that developers now work much faster and produce better quality products but only give feedback on notation and symbols. This can be because the other parts of the DSM solution are not as visible to modelers, and notational issues are always easy targets for opinions.

A good practice when defining the language is to ask users to come up with the symbols and other representational elements. This increases user involvement, helps in solving any “not-invented-here” attitude, and makes the language easier to learn and use. By language user, we don’t mean only those creating the models. The more models are used for purposes other than designing for code generation, the more likely we need others’ involvement. Those who read models, for example, to validate the specifications could participate too. Also, if some models are used to make product presentations by sales people or create configuration tools for deployment, it is good to ask opinions from this larger audience. It is much easier to introduce them to something they helped define: it has become their language, not yours.

10.6.1 Selecting Notation and Representational Forms

For defining a notation several guidelines can be given. For choosing notation, Costagliola et al. (2002) present a framework of classification and Rozenberg 1997 has edited a handbook on graph grammars and graph computing. In general, we can use anything for notation, from photorealism to an abstract box. Based on studies on cognitive dimensions (Blackwell, 1998), implicit, stylized pictograms are often better than photorealism, making models easier to read, understand, remember and work with. Photorealism also makes it hard to add text within symbols. This calls for more abstract notational elements that also illustrate detailed information with textual or pictorial properties. The worst approach is to use the same abstract symbol for different concepts. Using the same symbol for all the different concepts is like trying to understand a foreign language where the only letter is A with 20 slight variations of inflection! This is unfortunately often the case when using profiles to modify class diagrams. All the different domain concepts look the same: a rectangle. The only difference is a stereotype label, which is often presented similarly to the rest of the design information: using the same font and location inside the main symbol that is used for the other central data in the model.

Selecting Symbols for the Notation The best practice is to take representations for the notation and its symbols directly from the representations of the domain concepts. The sources for the notation are often the same as for the actual concepts. In cases where the domain is related, for example, to user interfaces, the notation can be defined rather easily, as was done in the mobile application case presented in Chapter 8. Also, if the domain to be modeled includes physical structures or has already established representational forms, the definition of the notation is considerably easier. You may use symbols of the physical product, such as a valve or sensor, to denote the actual software behind these concrete product elements. Usually all notational elements cannot be fully derived from already known symbols, so the rest must be created for languages. However, there is no reason to create new notation just for the sake of having something new. Instead, you should use and borrow good representations already in common use. Use the base language and model of computation you started with to help and guide you. For example, in models describing behavior such as state machines, flow diagrams, activity diagrams, and so on, well-established notation for start and end states already exists: a dot and a circle containing a dot. If your language gives additional semantics (modeling concepts or their properties) for the start and stop states, you may well extend the representation. For example, we may have a stop state that ends the application and another kind of stop that just ends the current function but not the whole application. These alternatives should also be made visible in the model by giving them different notation.

Usually, notation definition starts from the main modeling concepts but it is also reasonable to start with those symbols that come most naturally, giving a style that forms a basis for notation for other concepts too. When inspecting the other language concepts, you may find that they can use similar representations. Using a traffic sign for one concept may lead to considering other traffic signs for other concepts. Since notation can be richer than just showing basic design information, it is often best to define first the main notation and later add extensions that indicate special information, such as having a consistent symbol element to indicate that a model element is not completely defined, has errors, is described in more detail in a submodel, is reused from a library, and so on. Remember that notation can help the

### NOTATION FOR THE LANGUAGE

reader and it can change based on the properties it has or even based on the connections it has.

Each concept of the modeling technique should normally have one identifying representation. You should find representations that differentiate the concepts. This principle minimizes the overload of notational constructs and guarantees that all domain concepts can be distinguished in the models. Accordingly, the completeness of representations (Batani et al., 1992; Venable, 1993) or representational fidelity (Weber and Zhang, 1996), that is, availability of exactly one notational construct for each concept, is a well-known criterion for dealing with interpretations between modeling concepts and notations. If you have a lot of similar kinds of concepts that are still different in the modeling language (e.g., have different rules, connections, properties) you may use the same shapes, colors, fonts, and so on to identify their kind. This approach was used, for example, in CPL (Chapter 5). You may also define notational elements so that they visually show the same aspect, subdomain, reuse, origin, or architectural role. For example, in the watch case (Chapter 9) colors are used to show the MVC (Model-View-Controller) architecture.

Selecting a Representational Form The concrete syntax, how we represent the models, is not limited to symbols. We can choose among different representational styles too. The example models in this book illustrate representations of different graphical modeling languages where the symbols and icons represent different modeling concepts. Sometimes a matrix or a table works better than diagrams. A matrix is especially good if connections between model elements are important. A matrix also scales better than a diagram since more information can be shown in the same space. With matrix representation we thus don’t often need to partition model elements into submodels or parallel models. Table representation is especially good in showing properties of modeling elements: a parameter table is a classic example. The challenge in a table is showing the dependencies among the elements.

The choice of the representational form can also depend on the model manipulation actions. For example, a matrix gives a basis to automatically identify high cohesion and low coupling between model elements with diagonalization. Tables and matrices also offer other model manipulation options, like sorting based on model information. A classic example is finding priorities based on the properties of model elements. Diagrams are particularly good in finding patterns and organizing model elements into subdesigns. Ideally, we don’t need to fix on one representational style. The same model can be shown in different representational forms and model manipulation operations could take place in any of the possible representations.

10.6.2 Symbol Definition Guidelines

Although each notation will look different depending on the domain, some principles can be generalized. We have found the following symbol definitions to be effective.

. Use different kinds of notational elements for different modeling concepts.

. Use square and rectangle symbols when you need to show more text inside a symbol: the space can be used better than the ellipse, cloud, circle, triangle, and so on.

. Use vector graphics when the symbol needs to be scaled.

. Show only relevant data directly in the visual representation. Those days are gone when all the models were created with pen and paper showing all information on the same sheet. Today modeling tools can use filters and show the details in additional property sheets, dialogs, and browsers next to models.

. When the language is new for users, you may offer more guidance as part of the notation, like showing the name of the modeling concept as part of the symbol. Later, when the language has been learnt, you may remove them from the language definition or perhaps the tool allows users to hide them.

. Use colors. We hardly ever develop software that uses just black and white in its user interface, so why should models? Colors help in representing and reading the models and simply make them look better. You can use coloring and shading to illustrate different aspects or views, like MVC architecture (Chapter 9), similar kinds of domain concepts (Chapter 5), and UI (Chapter 8). Traffic light colors can be used to indicate preferences or choices, and different colors can be used to highlight priorities or follow the already established color use in the domain. Printers today can reproduce colored models with good results and modeling tools can also help in translating colors and different color settings into printable forms.

. Special effects such as small icons, shading, and fountain fills make models look better on screen but they don’t necessarily look as nice in other exported picture formats or printouts. They also make files used for interchanging models between tools bigger. Some interchange formats make the files for saving representational data unnecessarily large by saving every model element individually although their representational definition could be defined only once in the metamodel.

. In DSM languages targeting GUI or physical products you may apply two different notational versions of the same element: a 1:1 mapping to the realworld representation, or if that is not yet available, because no UI or product has yet been made, a more abstract notation.

. If different levels of detail need to be shown visually, you may apply different versions of the symbol based on user request, for example, one showing just a few properties and one showing complete details.

. If the notation aims to follow the real product closely, such as the GUI functionality described in Chapter 8 for the mobile phone case, different colors, fonts, and so on can be used to distinguish other model data from elements that resemble the real product. For example, the Multiquery object type in Chapter 8 uses gray for variable names to indicate that they are not part of the visible UI.

. Make symbols of model elements that may contain other elements transparent to allow showing parts inside the main symbol. The alternative is to create two

### TESTING THE LANGUAGES

symbols for the same aggregate concept: one showing information when the concept is not used as an aggregate and another when other concepts are part of the element (moved inside the aggregate symbol).

. Use consistent notational elements to improve model readability. For example, a special icon or text element can indicate if a model element is described in more detail or from another aspect in another model.

. You may consider using special notational elements, such as an icon or a text element, to illustrate the status of the model or of its individual model elements. This typically includes showing an error label for an incorrect model in cases where it was not feasible to include the error checking rules in the metamodel. In the same way, notation can indicate missing properties, showing the model is incomplete. It is generally better to place these special notational elements close to the model element that has the error or missing information to give a context. Textual model-wide checking reports are, however, better for more complex cases since they allow giving an explanation for the error or even instructions to correct the specification in a model. A hybrid solution— requiring more work—is to have both: a special notational element indicating the error in model and a model checking report that explains in more detail the status of the model.

. You may also inspect your corporate documentation standards: applying their look-and-feel in the modeling language improves acceptance and makes the generated documents more readable. Examples of such guidelines are classification schemes for the status of the model, such as draft, and frozen.

. Finally, do not be afraid to ask for help in making the notation: good metamodelers are not necessarily good graphic designers.

10.7 TESTING THE LANGUAGES

It is always a good idea to have multiple concrete example cases to test the language early on. Actually, the whole language creation could be considered as incremental and test case driven: first, you define some of the language, then model a little, make some extensions to the language, model some more, and so on. During each iteration you should test the language. The size of the iteration can be the size of the test case, but ideally, if the tools support it, you should be able to immediately test even small changes of the metamodel. Early and frequent testing as a part of language creation is especially relevant when you are defining your first modeling languages as it minimizes the risks of going in the wrong direction. Test cases also enable user participation and learning more about the domain and its modeling.

Testing of the modeling language is best done using examples from the real world—usually, the more cases the better. Ideally the cases should use as many modeling concepts as possible. At first the cases can be small features of a larger application and over time they are extended to cover the whole target of the language. You may use for two major approaches for finding test cases: rebuilding already developed applications or creating test cases from scratch just for language testing purposes. When testing is done by people other than the language developers, it is better to use known application features. You could even ask the developers who originally developed the application to test the defined language for the same application. Developers can then focus not on learning the application but on actual modeling. If generators of code, simulation, testing, and so on, are already available for the language, developers have extra motivation as models finally serve some other purpose than just documenting designs.

A large portion of the tests can be done by the language creators. This is natural, especially in the beginning, as the language is not necessarily complete, many changes may still be made, and multiple iterations can occur within a day. Later, when the modeling language is more complete, other developers should be involved. Their feedback is important not only for testing but also to smooth acceptance of the DSM solution.

At this stage, feedback is typically received about notation. Do the symbols look nice? As anybody can have an opinion on the notation, it is more relevant for DSM to focus on the language’s capabilities—conceptual structure instead of representation. It is often effective to ask language users to correct representational issues, such as how symbols should look, and in this way try to move testing to more substantial issues. Another good practice is to gather experiences while modeling by including in the language a special comment element: modelers may thus highlight issues they found relevant, but that could not be described with the language, and so on.

There are not many publications on testing and validating modeling languages (e.g., Fitzgerald, 1991; Schipper and Joosten, 1996. While testing the language, we have found three issues to be relevant (Tolvanen, 1998).

1. Abstraction: expressive power when describing the problem domain
2. Model consistency: organizing and keeping models consistent, guiding onreuse
3. Support for generators: producing required output from models, typically thecode

(1) Abstraction deals with comparing how well the given test case or application domain in general can be captured with the defined language. Modelers easily get back to you if the expressiveness is not adequate, but you may also check issues like

. Could the abstraction be higher? If the same kinds of patterns or combinations of model elements occur often, they should be replaced with a new modeling concept. Then, there is less to model and reuse of good practices is easier as the language provides concepts for them. . Does the language minimize the modeling effort? If there are repeating elements used just to make the model complete, these could be left to the

### TESTING THE LANGUAGES

generator. For example, in the mobile phone case (Chapter 8) almost half of the navigation flows can be removed in typical applications by moving standard cancel navigation to the generator.

. Have modelers made their own extensions? Models including special naming policies, such as prefix, postfix, capital letters, and special characters, can indicate that modeling power is not adequate. You should find out what is behind this special naming. Sometimes, the naming can be done just to satisfy the naming policy for the generated target language, keeping the generator simpler. If the modelers are not programmers, it is usually, better to let the generator take care of naming, for example, removing spaces from entries in models that are used for variable naming. . Are all modeling concepts needed? If the language includes concepts that are not applied, those could be removed as they add an extra burden for language users. Sometimes, during early phases of language creation there is a tendency to try to cover more of the domain than is actually needed. Finally getting a chance to master the modeling and code generation, you may go a bit too far.

1. Model consistency deals with how well the models are kept “correct.” . Do the specifications follow the language rules? Did we prevent making false specifications or specifications that would lead to poor performance? While metamodel-based rules should be naturally followed in a modern DSM tool, they are not necessarily specified correctly in the first place. Also rules that are supported by separate checks might not be followed at all. You may make checking mandatory by relating it to major modeling actions, forcing it to be run before versioning the model, or checking models while generating code, documentation, or other artifacts.

. How well is refactoring supported? If an element is changed in one model, are modelers forced to update other models manually? Could the same language, or different integrated languages, share concepts? In Chapter 8, the concept of a return variable is applied as the message element of an SMS sending object. Thus, if the name of thevariable is changed in any model element and it is used in SMS sending as a message element, the update is automatic. There is no need for the modeler to update the SMS sending element. You can find a similar integration structure, for example, in the button concept in the watch example (Chapter 9). If the button definition is changed in the watch display specification, the change is automatically reflected in all those applications, where the changed button is used—even though they are based on a different language. Today, when projects have multiple developers and they reuse others’ work, this capability saves time and prevents errors that are otherwise easy to make.

. Did the language support reuse? Are all the modelers creating similar kinds of models instead of using existing models? You may detect the patterns among models and update the language so it covers those patterns of reuse. Try to make it so model and even element boundaries fall in places that make natural units for reuse.

. Do we have rules that are too strict? Sometimes there is a tendency, perhaps because rules were easy to identify, to define language so strictly as to actually prevent modeling actions. You may, for example, correctly define the rules but they make sense only when the models are ready and remove freedom during early sketching or when making model modifications. A typical case here is when model elements can’t be changed but need to be deleted before a new correct model element can be defined. As the temporary deletion may also delete related properties, relationships, and even submodels, modelers are forced to model part of the specification twice. Here, checking the consistency rules only on request instead of at modeling time usually solves the problem.

1. Automation with generators normally deals more with the capability of the generator than with a modeling language. However, sometimes certain generation actions simply can’t be implemented since the initial input is not available or is not easily accessible from the models. Here, you should analyze examples that combine the original data in models and their implementation code as produced by the generators. We will discuss generators in detail in the next chapter.

10.8 MAINTAINING THE LANGUAGES

Languages do not remain static but evolve over time when the domain or generation needs change or when modelers, after having learned to use the language, see new opportunities for modeling. Whatever the change, it is usually best done in a similar manner to how the languagewasdeveloped:test with sample models to seetheinfluence. If the extension is large, changing many parts of the metamodel, then it is best is to do a pilot study before introducing the language updates. The whole creation process of a DSM solution, of which the modeling language is a part, is discussed in Chapter 13.

Here we inspect changes to the abstract syntax, which we specify in the metamodel. Changes in the concrete syntax, the notation, should always be easy to do and reflect in the models. During maintenance you may then add, modify, or delete any of the previously defined modeling concepts.

10.8.1 Adding New Concepts

In general, adding new concepts to modeling languages is easy. After having updated the generator and possibly the domain framework, modelers can continue development—by creating new models or updating the old ones. For example, after using the modeling language to develop mobile phone applications (Chapter 8),

### MAINTAINING THE LANGUAGES

the underlying target environment and its API changed, adding support for threads: When an external application is called for browsing the web or showing images from the file system, another process can be initiated instead of running it in a single thread. This change was put in the language as a modeling concept and no further actions were needed: new applications could then use separate processes. If an existing model wants to use the functionality provided by the new concept, it can be done naturally with a small change to the model.

If the added concept changes rules and makes some existing models illegal, you must decide how to change the models, if at all. Often no model update is needed here since the models are still usable: the generator still produces working codewith the old models. If you prefer that models are updated to follow the new constraint, the most typical decision is to inform modelers on those parts in the models that have become invalid. They can then see the model context and decide on actions for their update. If the added rule is well bounded and the model update policy is easy to formulate, you can use tools to define the model update transformation.

10.8.2 Removing Modeling Concepts

Removing and modifying existing concepts needs to be done more carefully: most likely these changes will have an effect on existing models. Good modeling tools should allow using existing models even if some elements of the metamodel are removed.

When the language is modified, the easy way out is to freeze the current language and continue its use only for application maintenance purposes. This is more likely an option if the target environment and domain framework are frozen too. It is more typical, however, that the project wants to upgrade to the newer language. Your task as the language definer is then to analyze in detail the possible side effects: how other data is modified too. It may be that removal of a concept has consequences and may remove more model elements than intended. For example, removing an object type may cause removal of some relationships too.

When after changing a metamodel, it is necessary to test the changes with real model data. After releasing the new version of the language, it is good practice to notify modelers of where the change has happened. This allows them to find the models that may need to be updated. Usually, the update can’t be fully automated as the change needed in the model depends on the context, which the model creators know. Depending on the tool, the notification can be made available using model browsers or running a generator that reports on all the models where the change could be made. Depending on the change, it can also be the case that no further notification is needed: when the developers change the models, they will be updated gradually to the new metamodel.

While considering the possible changes you can also identify which changes need the most time. For example, if you need to delete an existing concept and replace it with two new ones, you can consider just refactoring the old one so that the model changing work can be minimized. Naturally the old one is modified, if possible, to resemble the more common concept in models. A DSM tool then makes the update at least partly automated.

10.9 SUMMARY

The goal of defining a domain-specific language is to provide the software modelers and developers with a higher level language with which they can build systems. The best advice is to forget the implementation and code structures, at least in the beginning, and think about the problem domain. This raises the abstraction most and leads to fundamental productivity improvements. If you focus on the code from the start, the possibilities for automation may be easier to define but the gains are small, 10–30% improvements on current manual practices.

While it makes sense to develop the whole DSM solution at the same time, language definition can be started even when other elements of the DSM solution, or even the target environment for the resulting software, are not yet known. Modelers can then start development work, while others implement the supporting target environment, its libraries, and components along with code generators.

When identifying the modeling concepts, it is of key importance to focus on a narrow application domain and your actual needs for it, knowing that you can change the language when your requirements change. This support for language evolution is essential when it comes to making a choice of what DSM tooling to choose. Good environments allow such evolution, automatically updating all the models created previously with the language, whereas with less mature environments you can end up having to freeze or rebuild your models.