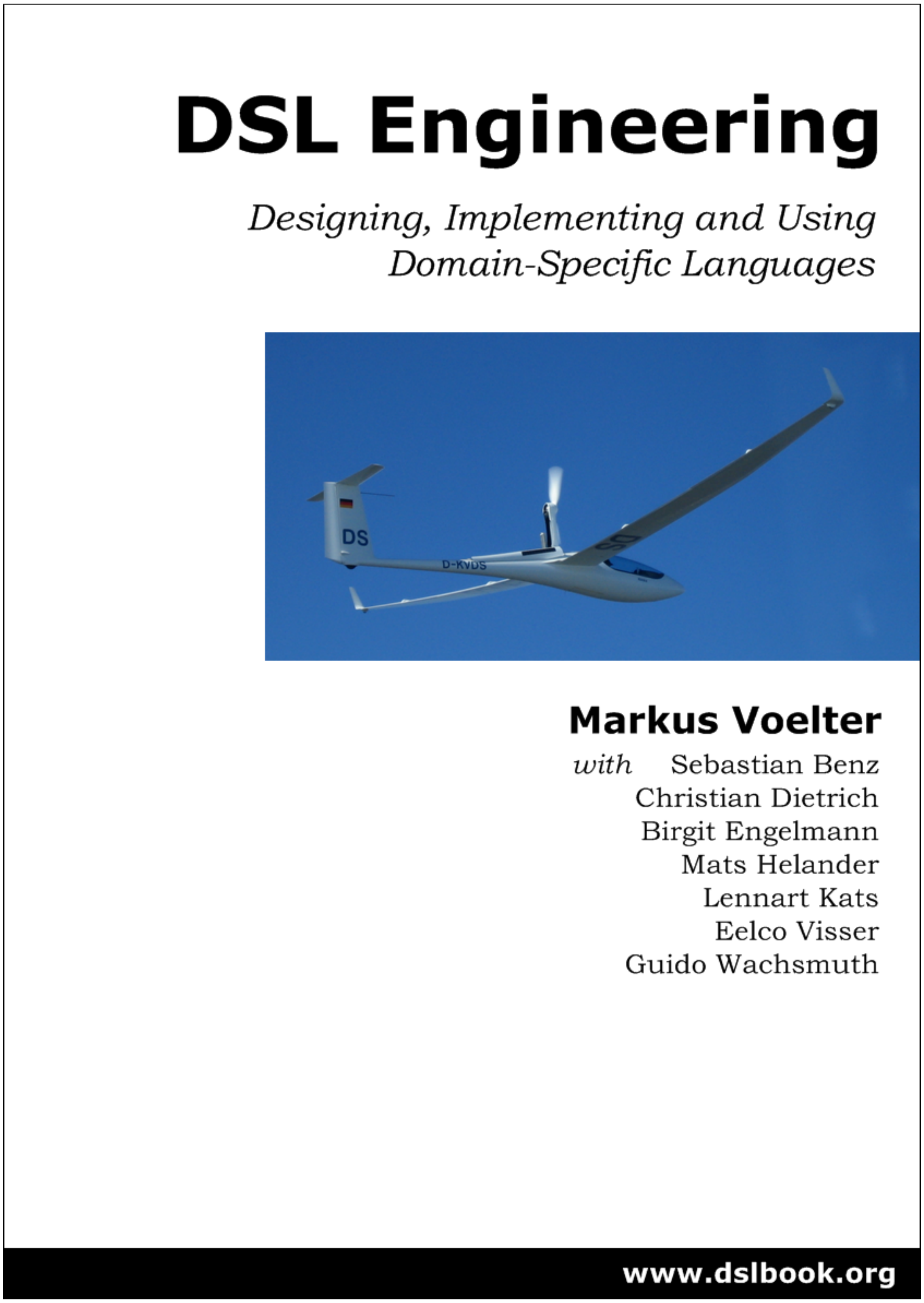


DSL Engineering

Designing, Implementing and Using Domain-Specific Languages

Markus Voelter
with Sebastian Benz
Christian Dietrich
Birgit Engelmann
Mats Helander
Lennart Kats
Eelco Visser
Guido Wachsmuth

www.dslbook.org

Part I

DSL Design

This part of the book has been written together with Eelco Visser of TU Delft. Reach him via `e.visser@tudelft.nl`.

Throughout this part of the book we refer back to the five case studies introduced in Part I of the book (Section 2.1). We use a the following notation:

Component Architecture: This refers to the component architecture case study described in Section ??. ◀

Refrigerators: This refers to the refrigerator configuration case study described in Section ??. ◀

Embedded C: This refers to the mbeddr.com extensible C case study described in Section ??. ◀

Pension Plans: This refers to the pension plans case study described in Section ??. ◀

WebDSL: This refers to the WebDSL case study described in Section ??. ◀

Note that, in this part of the book, the examples will only be used to illustrate DSL *design* and the driving design decisions. Part II of the book will then discuss the implementation aspects.

Some aspects of DSL Design have been formalized with mathematical formulae. These are intended as an additional means of explaining some of the concepts. Formulae are able to state properties of programs and languages in an unambiguous way. However, I want to emphasize that reading or understanding the formulae is *not* essential for understanding the language design discussion. So if you're not into mathematical formulae, just ignore them.

The part consists of three chapters. In Section 1 we introduce important terms and concepts including domain, model purpose and the structure of programs and languages. In Section 2 we discuss a set of seven dimensions that guide the design of DSLs: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax. Finally, in Section 3 we look at well-known structural and behavioral paradigms (such as inheritance or state based behaviour) and discuss their applicability to DSLs.

1 Conceptual Foundations

This chapter provides the conceptual foundations for the discussion of the design dimensions. It consists of three sections. The first one, Program, Languages and Domain *defines some of the terminology around DSL design we will use in the rest of this chapter. The second section briefly address the* Purpose *of programs as a way of guiding their design. And the third section briefly introduces parser-based and projectional editing, since some design considerations depend on this rather fundamental difference in DSL implementation.*

1.1 Programs, Languages and Domains

Domain-specific languages live in the realm of *programs*, *languages*, and *domains*. So we should start out by explaining what these things are. We will then use these concepts throughout this part of the book.

As part of this book's treatment of DSLs, we are primarily interested in *computation*, i.e. we are aimed at creating executable software[1]. So let's first consider the relation between programs and languages. Let's define P to be the set of all conceivable programs. A *program* p in P is the *conceptual* representation of some *computation* that runs on a universal computer (Turing machine). A *language* l defines a structure and notation for *expressing* or *encoding* programs from P. Thus, a program p in P may have an expression in L, which we will denote as p_l.

[1] This is opposed to just communicating among humans or describing complete systems.

There can be several languages l_1 and l_2 that express the *same* conceptual program p in different way p_{l_1} and p_{l_2} (`factorial` can be expressed in Java and Lisp, for example). There may even be multiple ways to express the same program in a single language l (in Java,

factorial can be expressed via recursion or with a loop). A transformation T between languages l_1 and l_2 maps programs from their l_1 encoding to their l_2 encoding, i.e. $T(p_{l_1}) = p_{l_2}$.

Notice that this transformation only changes the language used to express the program. The conceptual program does not change. In other words, the transformation preserves the semantics of p_{l_1}. We will come back to this notion as we discuss semantics in more detail in Section 2.3.

It may not be possible to encode all programs from P in a given language l. We denote as P_l the subset of P that can be expressed in l. More importantly, some languages may be *better* at expressing certain programs from P: the program may be shorter, more readable or more analyzable.

Turing-complete languages can by definition express all of P

Pension Plans: The pension plan language is very good at representing pension calculations, but cannot practically be used to express general purpose enterprise software. For example, user defined data structures and loops are not supported. ◀

■ *Domains* What are domains? We have seen one way of defining domains in the previous paragraph. When we said that a language l covers a subset of P, we can simply call this subset the *domain* covered with l. However, this is not a very useful approach, since it equates the scope of a domain trivially with the scope of a language (the subset of P in that domain P_D is equal to the subset of P we can express with a language l P_l). We cannot ask questions such as: "does the language adequately cover the domain?", since it always does, by definition.

There are two more useful approaches. In the *inductive* or *bottom-up* approach we define a domain in terms of existing software used to address a particular class of problems or products. That is, a domain D is identified as a set of programs with common characteristics or similar purpose. Notice how at this point we do *not* imply a special language to express them. They could be expressed in any Turing complete language. Often, such domains do not exist outside the realm of software. An especially interesting case of the inductive approach is where we define a domain as a subset of programs written in a specific language P_l instead of the more general set P. In this case we can often clearly identify the commonalities among the programs in the domain, in the form of their consistent use of a set of domain-specific patterns or idioms[2]. This makes building a DSL for D relatively simple, because we know exactly what the DSL has to cover, and we know what code to generate from DSL programs.

[2] Some people have argued for a long time that the need to use idioms or patterns in a language is a smell, and should be understood as hints at missing language features: `http://bit.ly/715wiF`

Embedded C: The domain of this DSL has been defined bottom-up. Based on idioms commonly employed when using C for embedded software development, linguistic abstractions have been defined that provide a "shorthand" for those idioms. These linguistic abstractions form the basis of the language extensions. ◀

The above examples can be considered relatively general – the domain of embedded software development is relatively broad. ~~In contrast,~~ a domain may also be very specific.

Refrigerators: The cooling DSL is tailored specifically at expressing refrigerator cooling programs for a very specific organization. No claim is made for broad applicability of the DSL. However, it perfectly fits into the way cooling algorithms are described and implemented in that particular organization. ◀

The second approach for defining a domain is *deductive* or *top-down*. In this approach, a domain is considered a body of knowledge about the real world, i.e. outside the realm of software. From this perspective, a domain D is a body of knowledge for which we want to provide some form of software support. P_D is the subset of programs in P that implement interesting computations in D. This case is much harder to address using DSLs, because we first have to understand precisely the nature of the domain and identify the interesting programs in that domain.

Pension Plans: The pensions domain has been defined this way. The customer had been working in the field of old age pensions for decades and had a very detailed understanding of what the pension domain entails. That knowledge was mainly contained in the heads of pension experts, in pension plan requirements documents, and, to a limited extent, encoded in the source of existing software. ◀

In the context of DSLs, we can ultimately consider a domain D by a set of programs P_D, whether we take the deductive or inductive route. There can be multiple languages in which we can express P_D programs. Possibly, P_D can only be partially expressed in a language l (Figure 1.1).

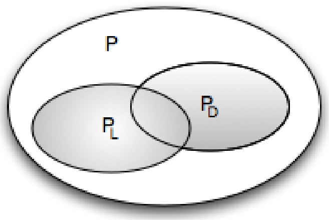


Figure 1.1: The programs relevant to a domain P_D and the programs expressible with a language P_L are both subsets of the set of all programs P. A good DSL has a large overlap with its target domain ($P_L \approx P_D$).

■ *Domain-Specific Languages* We can now understand the notion of a domain-specific language. A *domain-specific language* l_D for a domain D is a language that is *specialized* to encoding programs from P_D. That is, l_D is more efficient in representing P_D programs than other languages, and thus, is particularly well suited for P_D[3]. It achieves this by using *abstractions* suitable to the domain, and avoiding details that are irrelevant to programs in D (typically because they are similar in all programs and can be added automatically by the execution engine).

[3] There are several ways of measuring efficiency. The most obvios one is the amount of code a developer has to write to express a problem in the domain: the more concise, the more efficient. We will discuss this in more detail in Section 2.1.

It is of course possible to express programs in P_D with a general-purpose language. But this is less efficient – we may have to write much more code, because a GPL is not specialized to that particular domain. Depending on the expressivity of a DSL, we may also be able to use it to describe programs outside of D[4]. However, this is often not efficient at all, because, by specializing a DSL for D, we also restrict its efficiency for expressing programs outside of D. This is not a problem

[4] For example, you *can* write any program with some dialects of SQL.

as long as we have scoped D correctly. If the DSL actually just covers a subset of P_D, and we have to express programs in D for which the DSL is *not* efficient, we have a problem.

This leads us to the crucial challenge in DSL design: finding regularity in a non-regular domain and capturing it in a language. Especially in the deductive approach, membership of programs in the domain is determined by a human an is, in some sense, arbitrary. A DSL for the domain hence typically represents an explanation or interpretation of the domain, and often requires trade-offs by under- or over-approximation (Figure 1.2). This is especially true while we develop the DSL: an iterative approach is necessary that evolves the language as our understanding of the domain becomes more and more refined over time. In a DSL l that is adequate for the domain, the sets P_l and P_D are the same.

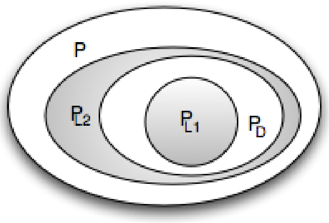


Figure 1.2: Languages L1 and L2 under-approximate and over-approximate domain D.

■ *Domain Hierarchy* In the discussion of DSLs and progressively higher abstraction levels it is useful to consider domains organized in a hierarchy[5], where higher domains are a subset (in terms of scope) of the lower domains (Fig. 1.3).

[5] In reality, domains are not always as neatly hierarchical as we make it seem in this section. Domains may overlap, for example. Nonetheless, the notion of a hierarchy is very useful to discuss many of the advanced topics in this book. In terms of DSLs, overlap may be addressed by factoring the common aspects into a separate language module that can be used in both of the overlapping domains.

At the bottom we find the most general domain D_0. It is the domain of all possible programs P. Domains D_n, with $n > 0$, represent progressively more specialized domains, where the set of interesting programs is a subset of those in D_{n-1} (abbreviated as D_{-1}). We call D_{+1} a subdomain of D. For example, $D_{1.1}$ could be the domain of embedded software, and $D_{1.2}$ could be the domain of enterprise software. The progressive specialization can be continued ad-infinitum in principle. For example, $D_{2.1.1}$ and $D_{2.1.2}$ are further subdomains of $D_{1.1}$: $D_{2.1.1}$ could be automotive embedded software and $D_{2.1.2}$ could be avionics software[6]

[6] At the top of the hierarchy we find singleton domains that consist of a single program (a non-interesting boundary case).

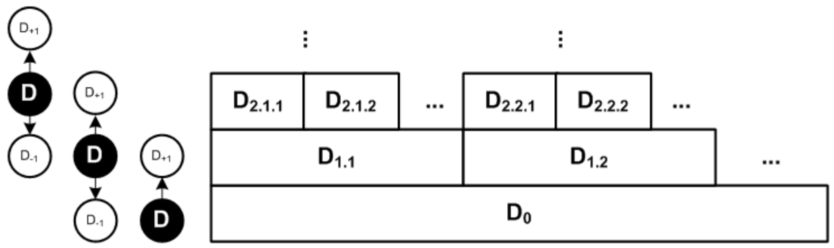


Figure 1.3: The domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

Languages are typically designed for a particular domain D. Languages for D_0 are called general-purpose languages[7]. Languages for D_n with $n > 0$ become more domain-specific for growing n. Languages for a particular D_n can also be used to express programs in D_{n+1}. However, DSLs for D_{n+1} may add additional abstractions or

[7] I guess we could define D_0 to be those programs expressible with Turing machines, but using GPLs for D_0 is a more useful approach for this book.

remove some of the abstractions found in languages for D_n. To get back to the embedded systems domain, a DSL for $D_{1.1}$ could include components, state machines and data types with physical units. A language for $D_{2.1.1}$, automotive software, will retain these extensions, but in addition provide direct support for the AUTOSAR standard. To conform to the MISRA-C standard, C's `void*` may be prohibited or removed.

Embedded C: The C base language is defined for D_0. Extensions for tasks, state machines or components can argued to be specific to embedded systems, making those sit in $D_{1.1}$. Progressive specialization is possible; for example, a language for controlling small Lego robots sits on top of state machines and tasks. It could be allocated to $D_{2.1.1}$. ◀

1.2 *Purpose*

We have said earlier that there can be several languages for the same domain. These languages differ regarding the abstractions they make use of. Deciding which abstractions should go into a particular language for D is not always obvious. The basis for the decision is to consider the *model purpose*. Models[8], and hence the languages to express them, are intended for a specific purpose. Examples of model purpose include automatic derivation of a D_{-1} program, formal analysis and model checking, platform independent specification of functionality or generation of documentation[9]. The same domain concepts can often be abstracted in different ways, for different purposes. When defining a DSL, we have to identify the different purposes required, and then decide whether we can create one DSL that fits all purposes, or create a DSL for each purpose[10].

[8] As we discuss below, we use the terms program and model as synonyms.

[9] Generation of documentation is typically not the main or sole model purpose, but may be an important secondary one. In general, we consider models that only serve communication among humans outside the scope of this book, because they don't have to be formally defined to achieve their purpose.

[10] Defining several DSLs for a single domain is especially useful if different stakeholders want to express different aspects of the domain with languages suitable to their particular aspect. We discuss this in the chapter on Viewpoints Section 2.4

Embedded C: The model purpose is the generation of an efficient low-level C implementation of the system, while at the same time providing software developers with meaningful abstractions. Since *efficient* C code has to be generated, certain abstractions, such as dynamically growing lists or runtime polymorphic dispatch are not supported even though they would be convenient for the user. The state machines in the `statemachines` language have an additional model purpose: model checking, i.e. proving certain properties about the state machines (e.g., proving that a certain state is definitely going to be reached after some event occurs). To make this possible, the action code used in the state machines is limited: it is not possible, for example, to read and write the same variable in the same action. ◀

Refrigerators: The model purpose is the generation of efficient implementation code for various different target platforms (different types of refrigerators use different electronics). A secondary purpose is enabling domain experts to express the algorithms and experiment with them using simulations and tests. The DSL is not expected to be used to visualize the actual refrigerator device for sales or marketing purposes. ◀

Pension Plans: The model purpose of the pension DSL is to enable insurance mathematicians and pension plan developers (who are not programmers) to define complete pension plans, and to allow them to check their own work for correctness using various forms of tests. A secondary purpose is the generation of the complete calculation engine for the computing center and the website. ◀

The purpose of a DSL may also change over time. Consequently, this may require changes to the abstractions or notations used in the language. From a technical perspective, this is just like any other case of language evolution (discussed in Section 4).

1.3 *The Structure of Programs and Languages*

The discussion above was relatively theoretical, trying to capture somewhat precisely the inherently imprecise notion of domains. Let us now move into the field of language engineering. Here we can describe the relevant concepts in a much more practical way.

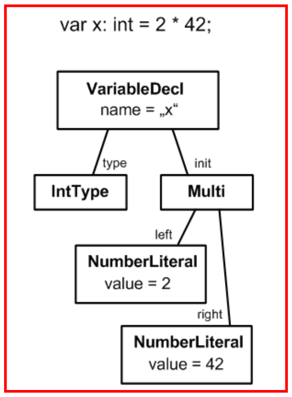


Figure 1.4: Concrete and abstract syntax for a textual variable declaration. Notice how the abstract syntax does not contain the keyword `var` or the symbols `:` and `;`.

The abstract syntax is very similar to a meta model in that it represents only a data structure and ignores notation. The two are also different: the abstract syntax is usually automatically derived from a grammar, whereas a meta model is typically defined *first*, independent of a notation. This means that, while the abstract syntax may be structurally affected by the grammar, the meta model is "clean" and represents purely the structure of the domain. In practice, the latter isn't strictly true either, since editing and tool considerations typically influence a meta model as well. In this book, we consider the two to be synonyms.

■ *Concrete and Abstract Syntax* Programs can be represented in their abstract syntax and the concrete syntax forms. The *concrete syntax* is the notation with which the user interacts as he edits a program. It may be textual, symbolic, tabular, graphical, or any combination thereof. The *abstract syntax* is a data structure that represents the semantically relevant data expressed by a program. It does not contain notational details such as keywords, symbols, white space or positions, sizes and coloring in graphical notations. The abstract syntax is used for analysis and downstream processing of programs. A language definition includes the concrete and the abstract syntax, as well as rules for mapping one to the other. *Parser-based* systems map the concrete syntax to the abstract syntax. Users interact with a stream of characters, and a parser derives the abstract syntax by using a grammar and mapping rules. *Projectional* editors go the other way round. User interactions, although performed through the concrete syntax, *directly* change the abstract syntax. The concrete syntax is a mere projection (that looks and feels like text in case a textual projection is used). No parsing takes place. Spoofax and Xtext are parser-based tools, MPS is projectional.

While concrete syntax modularization and composition can be a challenge and requires a discussion of textual concrete syntax details, we will illustrate most language design concerns based on the abstract syntax. The abstract syntax of programs ~~are primarily~~ trees of program *elements*. Each element is an instance of a *language concept*, or *concept* for short. A language is essentially a set of concepts (we'll come back to this below). Every element (except the root) is contained by exactly one parent element. Syntactic nesting of the concrete syntax corresponds to a parent-child relationship in the abstract syntax. There may also be any number of non-containing cross-references between elements, established either directly during editing (in projectional systems) or by a name resolution (or *linking*) phase that follows parsing and tree construction.

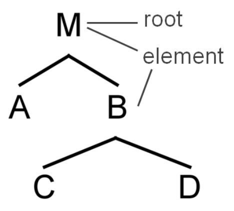


Figure 1.5: A program is a tree of program elements, with a single root element.

■ *Fragments* A program may be composed from several program *fragments*. A fragment is a standalone tree, a partial program. Conversely, a program is a set of fragments connected by references (discussed below). E_f is the set of program elements in a fragment f.

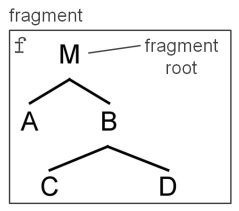


Figure 1.6: A fragment is an program tree that stands for itself and potentially references other fragments.

■ *Languages* A language l consists a set of language concepts C_l and their relationships[11]. We use the term *concept* to refer to concrete syntax, abstract syntax plus the associated type system rules and constraints as well as some definition of its semantics. In a fragment, each element e is an instance of a concept c defined in some language l.

[11] In the world of grammars, a concept is essentially a *Nonterminal*. We will discuss the details about grammars in the implementation section of this book

Embedded C: In C, the statement `int x = 3;` is an instance of the `LocalVariableDeclaration` concept. `int` is an instance of `IntType`, and the `3` is an instance of `NumberLiteral`. ◀

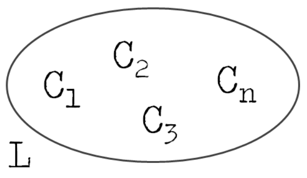


Figure 1.7: A language is a set of concepts.

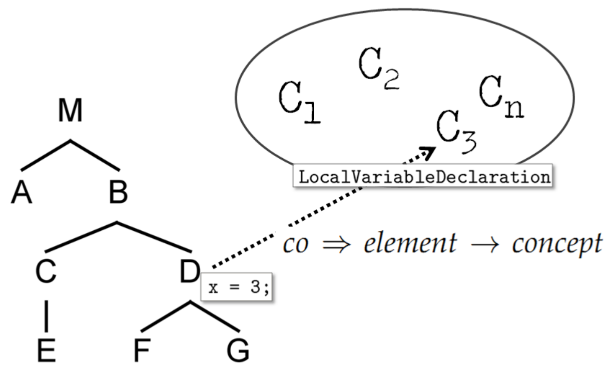


Figure 1.8: The statement `int x = 3;` is an instance of the `LocalVariableDeclaration`. *co* returns the concept for a given element.

■ *Functions* We define the *concept-of* function co to return the concept of which a program element is an instance: $co \Rightarrow element \rightarrow concept$ (see Fig. 1.8). Similarly we define the *language-of* function lo to return the language in which a given concept is defined: $lo \Rightarrow concept \rightarrow language$. Finally, we define a *fragment-of* function fo that returns the fragment that contains a given program element: $fo \Rightarrow element \rightarrow fragment$ (Fig. 1.9).

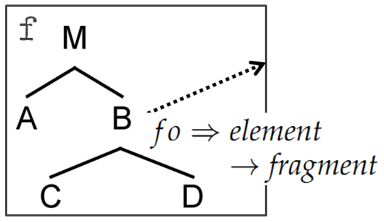


Figure 1.9: fo returns the fragment for a given element.

■ *Relations* We also define the following sets of relationships between program elements. Cdn_f is the set of parent-child relationships in a fragment f. Each $c \in Cdn$ has the properties $parent$ and $child$ (see figure Fig. 1.10; Cdn are all the parent-child "lines" in the picture).

Embedded C: In `int x = 3;` the local variable declaration is the *parent* of the `type` and the `init` expression `3`. The concept `LocalVariableDeclaration` defines the containment relationships `type` and `init`, respectively. ◀

$Refs_f$ is the set of non-containing cross-references between program elements in a fragment f. Each reference r in $Refs_f$ has the properties $from$ and to, which refer to the two ends of the reference relationship (see figure Fig. 1.10).

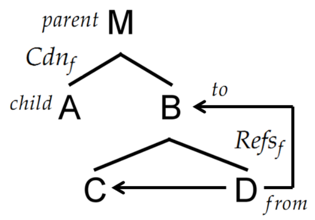


Figure 1.10: fo returns the fragment for a given element.

Embedded C: For example, in the `x = 10;` assignment, `x` is a reference to a variable of that name, for example, the one declared in the previous example paragraph. The concept `LocalVariableRef` has a non-containing reference relationsip `var` that points to the respective variable. ◀

Finally, we define an inheritance relationship that applies the Liskov Substitution Principle (LSP) to language concepts. The LSP states that,

in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substitutes for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)

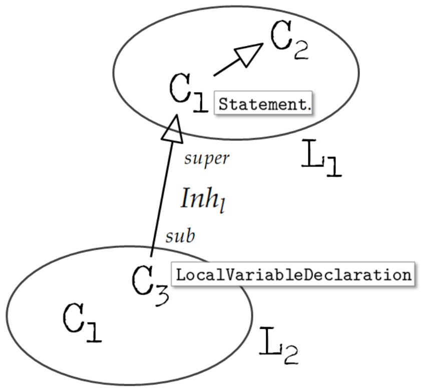


Figure 1.11: Concepts can extend other concepts. The base concept may be defined in a different language.

The LSP is well known in the context of object-oriented programming. In the context of language design it implies that a concept c_{sub} that extends another concept c_{super} can be used in places where an instance of c_{super} is expected. Inh_l is the set of inheritance relationships for a language l. Each $i \in Inh_l$ has the properties $super$ and sub.

Embedded C: The `LocalVariableDeclaration` introduced above extends the concept `Statement`. This way, a local variable declaration can be used wherever a `Statement` is expected, for example, in the body of a function, which is a `StatementList`. ◀

■ *Independence* An important concept is the notion of independence.

An *independent language* does not depend on other languages. This means that for all parent/child, reference and inheritance relationships, both ends refer to concepts defined in the same language. Based on our definitions above we can define an independent language l as a language for which the following hold[12]:

$$\forall r \in Refs_l \mid lo(r.to) = lo(r.from) = l \tag{1.1}$$

$$\forall s \in Inh_l \mid lo(s.super) = lo(s.sub) = l \tag{1.2}$$

$$\forall c \in Cdn_l \mid lo(c.parent) = lo(c.child) = l \tag{1.3}$$

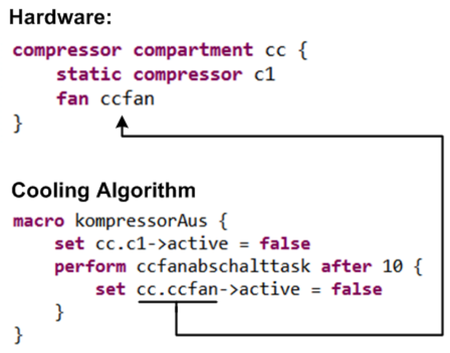
[12] As I said in the introduction to this part of the book, formulae like these are not essential for understanding. You may ignore them if you like.

Independence can also be applied to fragments. An *independent fragment* is one where all non-containing cross-references $Refs_f$ point to elements within the same fragment:

$$\forall r \in Refs_f \mid fo(r.to) = fo(r.from) = f \tag{1.4}$$

Notice that an independent language l can be used to construct dependent fragments, as long as the two fragments just contain elements from this single language l. Vice versa, a dependent language can be used to construct independent fragments. In this case we just have to make sure that the non-containing cross references are "empty" in the elements in fragment f.

Refrigerators: The hardware definition language is independent, as are fragments that use this language. In contrast, the cooling algorithm language is dependent. The **BuildingBlockRef** concept declares a reference to the **BuildingBlock** concept defined in the hardware language (Fig. 1.12). Consequently, if a cooling program refers to a hardware setup using an instance of **BuildingBlockRef**, the fragment becomes dependent on the hardware definition fragment that contains the referenced building block. ◀



Hardware:

```
compressor compartment cc {
    static compressor c1
    fan ccfan
}
```

Cooling Algorithm

```
macro kompressorAus {
    set cc.c1->active = false
    perform ccfanabschalttask after 10 {
        set cc.ccfan->active = false
    }
}
```

Figure 1.12: A **BuildingBlockRef** references a hardware element from within a cooling algorithm fragment.

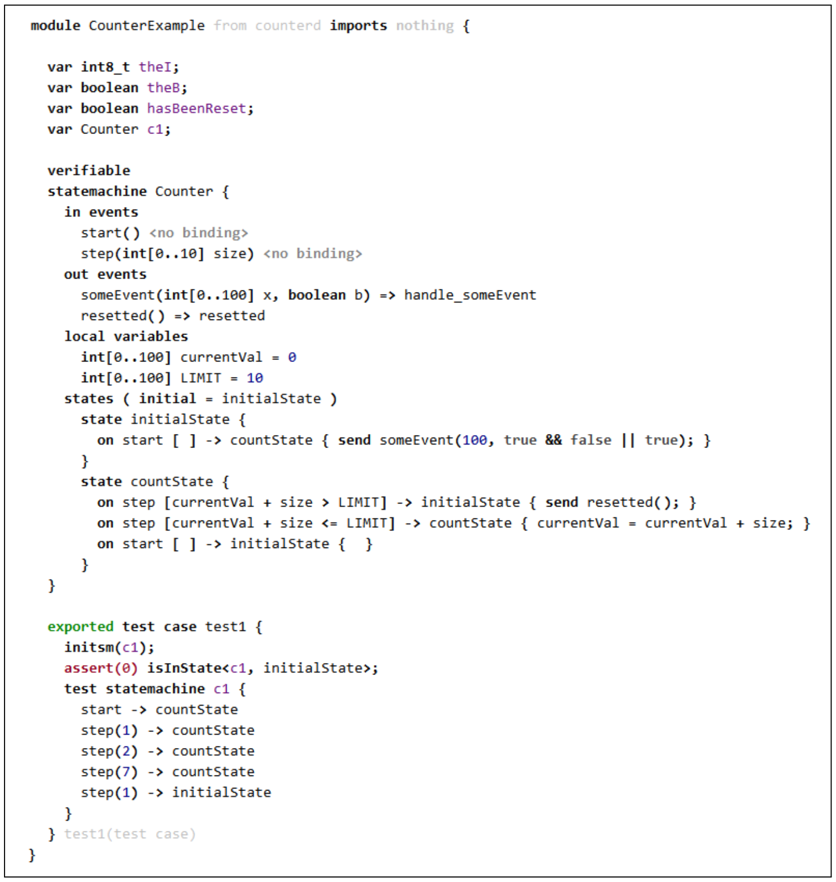
■ *Homogeneity* We distinguish *homogeneous* and *heterogeneous* fragments. A homogeneous fragment is one where all elements are expressed with the same language (see formula 1.5). This means that for all parent/child relationships (Cdn_f), the elements at both ends of the relationship have to be instances of concepts defined in one language l (1.6): [13]

$$\forall e \in E_f \mid lo(co(e)) = l \tag{1.5}$$

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = lo(co(c.child)) = l \tag{1.6}$$

[13] An independent language can only express homogeneous fragments. However, a homogeneous fragment can be expressed with a dependent language if the dependencies all come via the $Refs$ relationship.

Embedded C: A program written in plain C is homogeneous. All program elements are instances of the C language. Using the statemachine language extension allows us to embed state machines in C programs. This makes the respective fragment heterogeneous (see Fig. 1.13). ◀



```
module CounterExample from counterd imports nothing {

  var int8_t theI;
  var boolean theB;
  var boolean hasBeenReset;
  var Counter c1;

  verifiable
  statemachine Counter {
    in events
      start() <no binding>
      step(int[0..10] size) <no binding>
    out events
      someEvent(int[0..100] x, boolean b) => handle_someEvent
      resetted() => resetted
    local variables
      int[0..100] currentVal = 0
      int[0..100] LIMIT = 10
    states ( initial = initialState )
      state initialState {
        on start [ ] -> countState { send someEvent(100, true && false || true); }
      }
      state countState {
        on step [currentVal + size > LIMIT] -> initialState { send resetted(); }
        on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
        on start [ ] -> initialState {   }
      }
  }

  exported test case test1 {
    initsm(c1);
    assert(0) isInState<c1, initialState>;
    test statemachine c1 {
      start -> countState
      step(1) -> countState
      step(2) -> countState
      step(7) -> countState
      step(1) -> initialState
    }
  } test1(test case)
}
```

Figure 1.13: An example of a heterogeneous fragment. This module contains global variables (from the *core* language), a state machine (from the *statemachines* language) and a test case (from the *unittest* language). Note how concepts defined in the *statemachine* language (`trigger`, `isInState` and `test statemachine`) are used inside a `TestCase`.

1.4 *Parsing vs. Projection*

This part of the book is not about implementation techniques. However, the decision whether to build a DSL using a projectional editor instead of the more traditional parser-based approach can have some consequences for the design of the DSL. So we have to provide *some* level of detail on the two at this point.

In the parser-based approach, a grammar specifies the sequence of tokens and words that make up a structurally valid program. A parser is generated from this grammar. A parser is a program that recognizes valid programs in their textual form and creates an abstract syntax tree or graph. Analysis tools or generators work with this abstract syntax

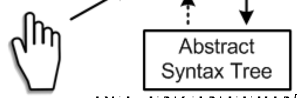


Figure 1.14: In parser-based systems, the user only interacts with the concrete syntax, and the AST is constructed from the information in the text.

tree. Users enter programs using the concrete syntax (i.e. character sequences) and programs are also stored in this way. Example tools in this category include Spoofax and Xtext.

Projectional editors (also known as structured editors) work *without* grammars and parsers. A language is specified by defining the abstract syntax tree, and then defining projection rules that render the concrete syntax of the language concepts defined by the abstract syntax. Editing actions *directly* modify the abstract syntax tree. Projection rules then render a textual (or other) representation of the program. Users read and write programs through this projected notation. Programs are stored as abstract syntax trees, usually as XML. As in parser-based systems, backend tools operate on the abstract syntax tree.

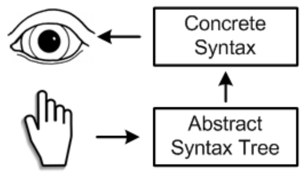


Figure 1.15: In projectional systems, the user sees the concrete syntax, but all editing gestures directly influence the AST. The AST is *not* extracted from the concrete syntax, which means the CS does not have to be parsable.

Projectional editing is well known from graphical editors, virtually all of them use this approach[14]. However, they can also be used for textual syntax[15], Example tools in this category include the Intentional Domain Workbench[16] and JetBrains MPS.

[14] You could argue that they are not *purely* projectional because the user can move the shapes around and the position information has to be persistent. Nonetheless, graphical editors are fundamentally projectional.

[15] While in the past projectional text editors have gotten a bad reputation mostly because of bad usability, as of 2011, the tools have become good enough, and computers have become fast enough to make this approach feasible, productive and convenient to use.

[16] `http://intentsoft.com`

In this section, we do not discuss the relative advantages and drawbacks of parser-based vs. projectional editors in any detail (we do discuss the trade-offs in the chapter on language implementation (Section **??**). However, we will point out if and when there are different DSL design options depending on which of the two approaches is used.

2 Design Dimensions

This chapter has been written ~~together~~ with Eelco Visser of TU Delft.
Contact him via `e.visser@tudelft.nl`.

DSLs are languages with high expressivity for a specific, narrow problem domain. They are powerful tools for software engineering, because they can be tailor-made for a specific class of problems. However, because of the large degree of freedom in designing DSLs, and because they are supposed to cover the right domain, consistently, and at the right abstraction level, DSL design is also hard. In this chapter we present a framework for describing and characterizing domain specific languages. We identify seven design dimensions that span the space within which DSLs are designed: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax. We illustrate the design alternatives along each of these dimensions with examples from our case studies. The dimensions provide a vocabulary for describing and comparing the design of existing DSLs and help guide the design of new ones. We also describe drivers, or forces, that lead to using one design alternative over another one. This chapter is not a complete methodology. It does not present a recipe that guarantees a great DSL if followed. I don't believe in methodologies, because they pretend precision where there isn't any. Building a DSL is a craft. This means that, while there are certain established approaches and conventions, building a good DSL also requires experience and practice.

2.1 Expressivity

One of the fundamental advantages of DSLs is increased expressivity over more general programming languages. Increased expressivity typically means that programs are shorter, and that the semantics are more readily accessible to processing tools (we will get back to this). By making assumptions about the target domain and encapsulating knowledge about the domain in the language and in its execution strategy (and not just in programs), programs expressed using a DSL can be significantly more concise.

Refrigerators: Cooling algorithms expressed with the cooling DSL are approximately five times shorter than the C version that users would have to write instead. ◀

While it is always possible to produce short but incomprehensible programs, in general, shorter programs require less effort to read and write than longer programs, and are therefore be more efficient in software engineering. We will thus assume that, all other things being equal, shorter programs are preferable over longer programs.[1]. We use the notation $|p_L|$ to indicate the size of program p as encoded in language L[2]. The essence is the assumption that, within one language, more complex programs will require larger encodings. We also assume that p_L is the smallest encoding of p in L, i.e. does not contain dead or convoluted code. We can then qualify the expressivity of a language relative to another language.

[1] The size of a program may not be the only relevant metric to asses the usefulness of a DSL. For example, if the DSL required only a third of the code to write, but it takes four times as long to write the code per line, the there is no benefit for writing programs. However, often when reading programs, less code is clearly a benefit. So it depends on the ratio between writing and reading code whether a DSL's conciseness is important.

[2] We will abstract over the exact way to measure the size of a program, which can be textual lines of code or nodes in a syntax tree, for example.

A language L_1 is *more expressive in domain D*
than a language L_2 ($L_1 \prec_D L_2$),
if for each $p \in P_D \cap P_{L_1} \cap P_{L_2}$, $|p_{L_1}| < |p_{L_2}|$.

A weaker but more realistic version of this statement requires that a language is *mostly* more expressive, but may not be in corner cases: DSLs may optimize for the common case and may require code written in a more general language to cover the corner cases[3].

[3] We discuss this aspect in the section on completeness (Section 2.5).

Compared to GPLs, DSLs (and the programs expressed with them) are more *abstract*: they avoid describing details that are irrelevant to the model purpose. The execution engine then fills in the missing details consistently, throughout the system in accordance with the knowledge about the domain encoded in the execution engine. Good DSLs are also *declarative*: they provide linguistic abstractions for relevant domain concepts that allow processors to "understand" the domain semantics without sophisticated analysis of the code. Linguistic abstraction means that a language contains concepts for the abstractions relevant in the domain. We discuss this in more detail below.

Note that there is a trade-off between expressivity and the scope of the language. We can always invent a language with exactly one symbol Σ that represent exactly one single program. It is extremely expressive! It is trivial to write a code generator for it. However, the language is also useless because it can only express *one single program*, and we'd have to create a new language if we wanted to express a different program. So in building DSLs we are striving for a language that has maximum expressivity while retaining enough coverage (see next chapter) of the target domain to be useful.

DSLs have the advantage of being more expressive than GPLs in the domain they are built for. But there is also a disadvantage: before being able to write these concise programs, users have to learn the language[4]. This task can be separated into learning the domain itself, and learning the syntax of the language. For people who know the domain, learning the syntax can be simplified by using good IDEs with code completion and quick fixes, as well as with good, example-based documentation. In many cases, DSL users already know the domain, or would have to learn the domain even if no DSL were used to express programs in the domain: learning the domain is independent of the language itself. It is easy to see, however, that, if a domain is supported by well-defined language, this can be a good reference for the domain itself. Learning a domain can be simplified by working with a good DSL[5]. In conclusion, the learning overhead of DSLs is usually not a huge problem in practice.

[4] While a GPL also has to be learned, we assume that there is a relatively small number of GPLs and developers already know it. ~~Using DSLs, there may be a larger number of them~~ used in any given project or organization, and new team members cannot be expected to know them.

[5] This can also be read the other way round: a measure for the quality of a DSL is how long it takes domain experts to learn it.

Pension Plans: The users of the pension DSL are pension experts. Most of them have spent years describing pension plans using prose text, tables and (informal) formulas. The DSL provides formal languages to express that same in a way that can be processed by tools. ◀

The close alignment between a domain and the DSL can also be exploited during the construction of the DSL. While it is not a good idea to start building a DSL for a domain about which we don't know much, the process of building the DSL can help deepen the understanding about a domain. The domain has to be scoped, fully explored and systematically structured to be able to build a language.

Refrigerators: Building the cooling DSL has helped the thermodynamicists and software developers to understand the details of the domain, its degrees of freedom and the variability in refrigerator hardware and cooling algorithms in a much more structured and thorough way than before. Also, the architecture of the to-be-generated C application that will run on the device ~~became~~ much more well-structured as a consequence of the separation between reusable frameworks, device drivers and generated code. ◀

2.1.1 *Expressivity and the Domain Hierarchy*

In the ~~definition of~~ expressivity above we compare arbitrary languages. An important idea behind domain-specific languages is that progressive specialization of the domain enables progressively more specialized and expressive languages. Programs for domain $D_n \subset D_{n-1}$ expressed in a language $L_{D_{n-1}}$ typically use a set of characteristic idioms and patterns. A language for D_n can provide linguistic abstractions for those idioms or patterns, which makes their expression much more concise and their analysis and translation less complex.

Embedded C: Embedded C extends the C programming language with concepts for embedded software including state machines, tasks, and physical quantities. The state machine construct, for example, has concepts representing states, events, transitions and guards. Much less code is required compared to **`switch/case`** statements or cross-indexed integer arrays, two typical idioms for state machine implementation in C. ◀

WebDSL: WebDSL entity declarations abstract over the boilerplate code required by the Hibernate framework for annotating Java classes with object-relational mapping annotations. This reduces code size by an order of magnitude [6]. ◀

6

2.1.2 *Linguistic vs. In-Language Abstraction*

There are two major ways of defining abstractions. Abstractions can be built into the language (in which case they are called *linguistic* abstraction), or they can be expressed by concepts available in the language (*in-language* abstraction). DSLs typically rely heavily on linguistic abstraction, whereas GPLs rely more on in-language abstraction.

■ *Linguistic Abstraction* A specific domain concept can be modeled with the help of existing abstractions, or one can introduce a *new* abstraction for that concept. If we do the latter, we use *linguistic* abstraction. By making the concepts of D first class members of a language L_D, i.e. by defining linguistic abstractions for these concepts, they can be uniquely identified in a D program and their structure and semantics is well defined. No semantically relevant[7] idioms or patterns are required to express interesting programs in D. Consider the two examples of loops in a Java-like language:

[7] By "semantically relevant" we mean that the tools needed to achieve the model purpose (analysis, translation) have to treat these cases specially.

```
int[] arr = ...
for (int i=0; i<arr.size(); i++) {
    sum += arr[i];
}
```

```
int[] arr = ...
OrderedList<int> l = ...
for (int i=0;  i<arr.size(); i++) {
     l.add( arr[i] );
}
```

The left loop can be parallelized, since the order of summing up the array elements is irrelevant. The right one cannot, since the order of the elements in the `OrderedList` class *is* relevant. A transformation engine that translates and optimizes the programs must perform (sophisticated, and sometimes impossible) program analysis to determine that the left loop can indeed be parallelized. The following alternative expression of the same behavior uses better linguistic abstractions, because it is clear without analysis that the first loop can be parallelized and the second cannot:

```
for (int i in arr) {              seqfor (int i in arr) {
    sum += i;                         l.add( arr[i] );
}                                 }
```

The property of a language L_D of having first-class concepts for abstractions relevant in D is often called *declarativeness*: no sophisticated pattern matching or program flow analysis is necessary to capture the semantics of a program (relative to the purpose), and treat it correspondingly. The decision can simply be based on the language concept used (`for` vs. `seqfor`)[8].

[8] Without linguistic abstraction, the processor has to analyze the program to "reverse engineer" the semantics to be able to act on it. With linguistic abstraction, we rely on the language user to use the correct abstraction. We assume that the user is able to do this! The trade-off makes sense in DSLs because we assume that DSL users are familiar with the domain, and we often don't have the budget or experience to build the sophisticated program analyses that could do the semantic reverse engineering.

Embedded C: State machines are represented with first class concepts. This enables code generation, as well as meaningful validation. For example, it is easy to detect states that are not reached by any transition and report this as an error. Detecting this same problem in a low-level C implementation requires sophisticated analysis on the switch-case statements or indexed arrays that constitute the implementation of the state machine[9] ◀

[9] This approach assumes that the generator works correctly – we'll discuss this problem in Section 2.3 on semantics).

Embedded C: Another good example is optional ports in components. Components (see Fig. 2.1) define required ports that specify the interfaces they *use*. For each component instance, a required port is connected to the provided port of an instance of a component that provides a port with a compatible interface. Required ports may be optional[10], so for a given instance, an optional port may be connected or not. Invoking an operation on an unconnected required port would result in an error, so this has to be prevented. This can be done by enclosing the invocation on a required port in an `if` statement, checking whether the port is connected. However, an `if` statement can contain any arbitrary Boolean expression as its condition (e.g., `if (isConnected(rp) || somethingRandom()) { port.doSomething(); }`). So checking *statically* that the invocation only happens if the port is connected is impossible. A better solution based on linguistic abstraction is to introduce a new language concept that checks for a connected port directly: `with port (rp) { rp.doSomething(); }`. The `with port` statement doesn't use an expression as its argu-

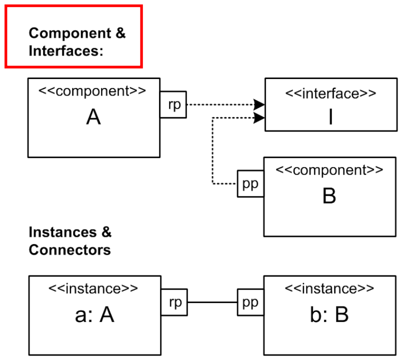
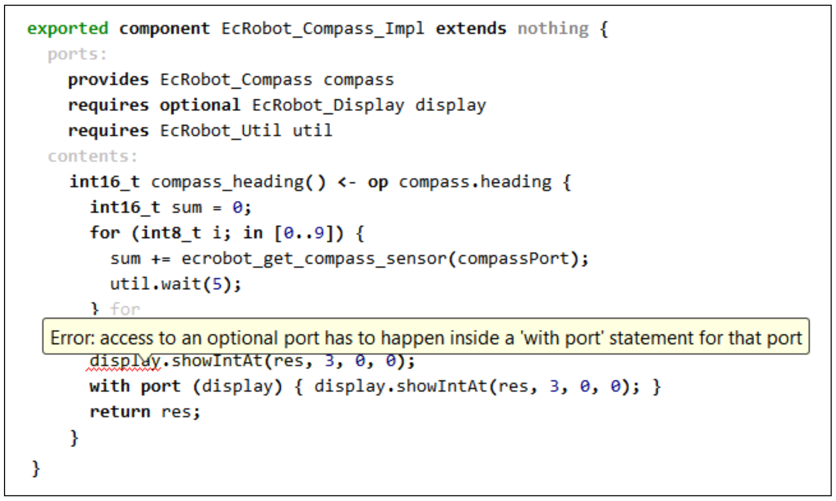


Figure 2.1: Example component diagram. The top half defines components, their ports and the relationship of these ports to interfaces. The bottom half shows instances whose ports are connected by a connector.

[10] The terminology may be a bit confusing here: *required* means that the component invokes operations on the port (as opposed to providing them for other to invoke). *optional* refers to the fact that, for any given *instance* of that component, the port may be connected or not.

ment, but only a reference to a an optional required port (Fig. 2.2). This way, the IDE can check that an invocation on a required optional port **`rp`** is only done inside of a **`with port`** referencing that same port. ◀



```
exported component EcRobot_Compass_Impl extends nothing {
  ports:
    provides EcRobot_Compass compass
    requires optional EcRobot_Display display
    requires EcRobot_Util util
  contents:
    int16_t compass_heading() <- op compass.heading {
      int16_t sum = 0;
      for (int8_t i; in [0..9]) {
        sum += ecrobot_get_compass_sensor(compassPort);
        util.wait(5);
      } for
  Error: access to an optional port has to happen inside a 'with port' statement for that port
      display.showIntAt(res, 3, 0, 0);
      with port (display) { display.showIntAt(res, 3, 0, 0); }
      return res;
    }
}
```

Figure 2.2: The **`with port`** statement is required to surround an invocation on an optional required port; otherwise, an error is reported in the IDE. If the port is not connected for any given instance, the code inside the **`with port`** is not executed. It acts like an **`if`** statement, but since it cannot contain an expression as its condition, the correct use of the **`with port`** statement can be statically checked.

Linguistic abstraction also means that no details irrelevant to the model purpose are expressed. Once again, this increases conciseness, and avoids the undesired specification of unintended semantics (overspecification). Overspecification is usually bad because it limits the degrees of freedom available to a transformation engine. In the example with the parallelizable loops, the fist loop is over-specified: it expresses ordering of the operations, although this is (most likely) not intended by the person who wrote the code.

Embedded C: State machines can be implemented as switch/case blocks or as arrays indexing into each other. The DSL program does not specify which implementation should be used and the transformation engine is free to chose the more appropriate representation, for example, based on desired program size or performance characteristics. Also, **`log`** statements and **`task`** declarations can be translated in different ways depending on the target platform. ◀

■ *In-Language Abstraction* Conciseness can also be achieved by a language providing facilities to allow users to define new (non-linguistic) abstractions in programs. Well-known GPL concepts for building new abstractions include procedures, classes, or functions and higher-order functions, generics, traits and monads It is *not* a sign of a bad DSL if it has in-language abstraction mechanisms as long as the created abstractions don't require special treatment by analysis or processing

It is worth understanding these to some extent, so you can make an informed decision which of these – if any – are useful in a DSL.

tools – at which point they should be refactored into linguistic abstractions. An example of such special treatment would be if the compiler of the above example language would know that the **`OrderedList`** library class is actually ordered, and that, consequently, the respective loop cannot be parallelized. Another example of special treatment can be constructed in the context of the optional port example. If ~~we'd solve~~ the problem by having a library function **`isConnected(port)`**, we could enforce ~~that a call on an optional port is surrounded~~ by an **`if (isConnected(port))`** *without any other expression* in the condition. In this case, the static analyzer would have to treat **`isConnected`** specially[11]. In-Language abstraction can, as the name suggests, provide *abstraction*. But it cannot provide *declarativeness*: a model processor has to "understand" what the user wanted to express by building the in-language abstraction, in order to be able to act on it.

[11] Treating program elements specially is dangerous because the semantics of **`isConnected`** or **`OrderedList`** could be changed by a library developer, without changing the static analyzer of code generator in a consistent way.

Refrigerators: The language does not support the construction of new abstractions since its user community are non-programmers who are not familiar with defining abstractions. As a consequence, the language had to be modified several times during development as new requirements came up from the end users, and had to be integrated directly into the language. ◀

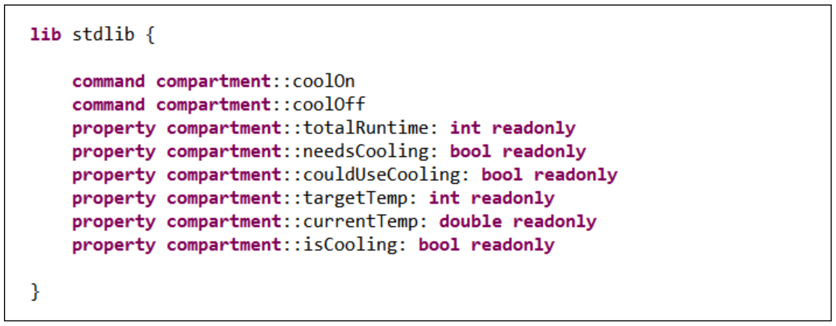
Embedded C: Since C is extended, C's abstraction mechanisms (functions, **`struct`**s, **`enum`**s) are available. Moreover, we added new mechanisms for building abstractions including interfaces and components. ◀

WebDSL: WebDSL provides *template definitions* to capture partial web pages including rendering of data from the database and form request handling. User defined templates can be used to build complex user interfaces. ◀

■ *Standard Library* If a language provides support for in-language abstraction, these facilities can be used by the *language developer* to provide collections of domain specific abstractions to language users. Instead of adding language features, a standard library is deployed along with the language to all its users. It contains abstractions relevant to the domain, expressed as in-language abstractions. This approach keeps the language itself small, and allows subsequent extensions of the library without changing the language definition and processing tools.

This approach is of course well known from programming languages. All of them come with a standard library, and the language can hardly be used without relying on it. It is effectively a part of the language

Refrigerators: Hardware building blocks have properties. For example, a **`fan`** can be turned **`on`** or **`off`**, and for a **`compressor`**, the speed can be specified (**`rpm`**). The set of properties available for the various building blocks is defined via a standard library and is not part of the language (Fig. 2.3). The reason why this is *not* a contradiction to what we discussed earlier is this: as a consequence of the structure of the framework used on the target platform, new properties can be added to hardware elements *without* the need to change the generator. They are not treated specially! ◄



```
lib stdlib {

    command compartment::coolOn
    command compartment::coolOff
    property compartment::totalRuntime: int readonly
    property compartment::needsCooling: bool readonly
    property compartment::couldUseCooling: bool readonly
    property compartment::targetTemp: int readonly
    property compartment::currentTemp: double readonly
    property compartment::isCooling: bool readonly

}
```

Figure 2.3: The standard library for the refrigerator configuration language defines which properties are available for the various types of hardware elements.

Some languages treat some abstractions defined in the standard library specially. For example, Java's `WeakReference` has special behavior in the context of garbage collection. While an argument can be made that special treatment is acceptable for a standard library (after all, it can be considered an essential companion to the language itself), it is still risky and dangerous. Considering that, in the case of DSLs, we can change the language relatively easily, I would suggest to avoid special treatment even in a standard library and recommend providing linguistic abstractions for these cases.

■ *Comparing Linguistic and In-Language Abstraction* A language that contains linguistic abstractions for all relevant domain concepts is simple to process; the transformation rules can be tied to the identities of the language concepts. It also makes the language suitable for domain experts, because relevant domain concepts have a direct representation in the language. Code completion can provide specific and meaningful support for "exploring" how a program should be written. However, using linguistic abstractions extensively requires that the relevant abstractions be known in advance, or frequent evolution of the language is necessary. It also can lead to languages that feel large, bloated or even inelegant. In-language abstraction is more flexible, because users can build just those abstractions they need. However, this requires that users are actually trained to build their own abstractions. This is often true for programmers, but it is typically not true for domain experts.

Using a standard library may be a good compromise where one set of users develops the abstractions to be used by another set of developers. This is especially useful if the same language should be used for several, related projects or user groups. Each can build their

Modular language extension, as discussed later in Section 2.6.2, provides a middle ground between the two approaches. A language can be flexibly extended, while retaining the advantages of linguistic abstraction.

own set of abstractions in the library.

Note that languages that provide good support for in-language abstraction feel different from those that use a lot of linguistic abstraction (compare Scala or Lisp to Cobol or ABAP). Make sure that you don't mix the two styles unnecessarily. The resulting language may be judged as being ugly, especially by programmers.

2.1.3 *Language Evolution Support*

If a language uses a lot of linguistic abstraction, it is likely, especially during the development of the language, that these abstractions change. Changing language constructs may break existing models, so special care has to be taken regarding language evolution. This requires any or all of the following: a strict configuration management discipline, versioning information in the models to trigger compatible editors and model processors, keeping track of the language changes as a sequence of change operations that can be "replayed" on existing models, or model migration tools to transform models based on the old language into the new language.

Whether model migration is a challenge or not depends on the tooling. There are tools that make model evolution a very smooth, but many environments don't. Consider this when deciding about the tooling you want to use!

In parser-based languages, you can always at the very least open the file in a text editor and run some kind of global search/replace to migrate the program. In projectional editor, special care has to be taken to enable the same functionality.

It is always a good idea to minimize those changes to a DSL that break existing models[12]. Backward-compatibility and deprecation are techniques well worth keeping in mind when working with DSLs. For example, instead of just changing an existing concept in an incompatible way, you ~~may~~ add a new concept in addition to the old one, along with deprecation of the old one and a migration script or wizard. Note that you might be able to instrument your model processor to collect statistics on whether deprecated language features continue to be used. Once no more instances show up in models, you can safely remove the deprecated language feature.

[12] This is especially true if you don't have access to all programs to migrate them in one fell swoop: you have to deploy migration scripts with the language or rely on the users to manually perform the migration.

If the DSL is used by a closed, known user community that is accessible to the DSL designers, it will be much easier to evolve the language over time because users can be reached, making them migrate to newer versions[13]. Alternatively, the set of all models can be migrated to a newer version using a script provided by the language developers. In case the set of users, and the DSL programs, are not easily accessible, much more effort must be put into keeping backward compatibility, the need for coordinated evolution should be reduced[14].

[13] The instrumentation mentioned above may even report back uses of deprecated language features after the official expiration date.

[14] This is the reason why many GPLs can never get rid of deprecated language features.

2.1.4 *Precision vs. Algorithm*

We discussed earlier that some DSLs may be Turing-complete (and feel more like a programming language) whereas others are purely declarative, and maybe just describe facts, structures and relationships in a domain. The former may not be usable by domain users (i.e. non-programmers). They are often able to formally and precisely specify facts, structures and relationships about their domain, but they are often not able to define algorithmic behavior.

In this case, a DSL has to be defined that abstracts far enough to hide these algorithmic details. Alternatively, you can create an incomplete language (Section 2.5), and have developers fill in the algorithmic details in GPL code. One way to do this is to provide a set of predefined behaviors (in some kind of library) which are then just parametrized or configured by the users.

Pension Plans: Pension rules are at the boundary between being declarative and algorithmic. The majority of the models define data structures (customers, pension plans, payment schedules). However, there are also mathematical equations and calculation rules. These are algorithmic, but in the pension insurance domain, the domain users are well capable of dealing with these. ◀

2.1.5 *Configuration Languages*

Configuration languages are purely declarative. They consist of a well-defined set of configuration parameters and constraints among them. "Writing programs" boils down to setting values for these parameters. In many cases, the parameters are Booleans, in which case a program is basically a selection of a subset of the configuration switches. A well-known configuration language is feature models. We discuss configuration languages in more detail in the chapter on DSLs and Product Line Engineering (Section **??**).

2.1.6 *Platform Influence*

In theory, the design of the abstractions used in a language should be independent of the execution engine and the platform. However, this is not always the case[15] There are two reasons why the platform may influence the language.

[15] It is obviously not the case for architecture DSLs where you build a language that resembles the architectural abstractions in a platform. But that's not what we're talking about here.

■ *Runtime Efficiency* In most systems, the resulting system has to execute in a reasonably efficient way. Efficiency can mean performance, scalability as well as resource consumption (memory, disk space, network bandwidth). Depending on the semantic gap between the platform and the language, building efficient code generators can be a lot of work (we discuss this in some detail in the ~~chapter on~~ semantics (Section 2.3)). Instead of building the necessary optimizers, you can also change the language to use abstractions that make global optimizations simpler to build. [16].

[16] While this may be considered "cheating", in may nonetheless be the only practical way given project constraints.

Embedded C: The language does not support dynamically growing lists because it is hard to implement them in an efficient way considering we are targeting embedded software. Dynamic allocation of memory is often now allowed, and even if it were, the necessary copying of existing list data into a new, bigger buffer is too expensive for practical use. The incurred overhead is also *not* obvious to the language user (he just increases list size or adds another element that triggers list growth), making it all the more dangerous. ◀

Embedded C: Another example includes floating point arithmetics. If the target platform has no floating point unit, floating point ~~arithmetics are~~ expensive to emulate (FPU). We had to build the language in a way that could prevent the use of **`float`** and **`double`** types if the target platform has no FPU. ◀

■ *Platform Limitations* The platform may have limitations regarding the size of data structures, the memory or disk space, or the bandwidth of the network that limit or otherwise influence language design.

Refrigerators: In the cooling language we had to introduce time units (seconds, minutes, hours) into the DSL after we'd noticed that the time periods relevant for cooling algorithms are so diverse, that no single unit could fit all necessary values into the available integer types. If we had used only seconds, the days or months periods would not fit into the available **`int`**s. Using only hours or days obviously would not let us express the short periods without using fractions of floating point data types. So the language now has the ability to express periods as in **`3s`** or **`30d`**. ◀

2.2 *Coverage*

A language L always defines a domain D such that $P_D = P_L$. Let's call this domain D_L, i.e. the domain determined by L. This does not work the other way around. Given a (deductively defined) domain D there is not necessarily a language that *fully covers* that domain unless we revert to a universal language at a D_0 (cf. the hierarchical structure of domains and languages).

Note that we can achieve full coverage by making L *too general*. Such a language, may, however, be less expressive, resulting in bigger (unnecessarily big) programs. Indeed this is the reason for designing DSLs in the first place: general purpose languages are too general.

A language L *fully covers* domain D, if for each program p relevant to the domain P_D a program p_L can be written in L. In other words, $P_D \subseteq P_L$.

Full coverage is a Boolean predicate; a language either fully covers a domain or it does not. In practice, many languages do not fully cover their respective domain. We would like to indicate the *coverage ratio*. The domain coverage ratio of a language L is the portion of programs in a domain D that it can express. We define $C_D(L)$, *the coverage of domain D by language L*, as

$$C_D(L) = \frac{number\ of\ P_D\ programs\ expressable\ by\ L}{number\ of\ programs\ in\ domain\ D}$$

At first glance, an ideal DSL will cover all of its domain ($C_D(L_D)$ is 100%). It requires, however, that the domain is well-defined and we can actually know what full coverage is. Also, over time, it is likely that the domain evolves and grows, and the language has to be continuously evolved to retain full coverage.

As the domain evolves, language evolution has to keep pace, requiring responsive DSL developers. This is an important process aspect to keep in mind!

In addition to the evolution-related reason given above, there are two reasons for a DSL *not* to cover all of its *own* domain D. First, the language may be deficient and needs to be redesigned. This is especially likely for new and immature DSLs. Scoping the domain for which to build a DSL is an important part of DSL design.

Second, the language may have been defined expressly to cover only a subset of D, typically the subset that is most commonly used. Covering all of D may lead to a language that is too big or complicated for the intended user community because of its support for rarely used corner cases of the domain[17]. In this case, the remaining parts of D may have to be expressed with code written in D_{-1} (see also Section 2.5). This requires coordination between DSL users and D_{-1} users, if this not the same group of people.

[17] Incremental language extension provides a third option: you can put the common parts into the base language and the support for the corner cases into optionally included language modules.

WebDSL: WebDSL defines web pages through "page definitions" which have formal parameters. `navigate` statements generate links to such pages. Because of this stylized idiom, the WebDSL compiler can check that internal links are to existing page definitions, with arguments of the right type. The price that the developer pays is that the language does not support free form URL construction. Thus, the language cannot express all types of URL conventions and does not have full coverage of the domain of web applications. ◀

Refrigerators: After trying to write a couple of algorithms, we had to add a `perform ...after t` statement to run a set of statements after a specified time `t` has elapsed. In the initial language, this had to be done manually with events and timers. Over time we noticed that this is a very typical case, so we added first-class support. ◀

Embedded C: Coverage of this set of languages is full, although any particular extension to C may only cover a part of the respective domain. However, even if no suitable linguistic abstraction is available for some domain concept, it can be implement in the D_0 language C, while retaining complete syntactic and semantic integration. Also, additional linguistic abstractions can be easily added because of the extensible nature of the overall approach. ◀

2.3 Semantics and Execution

Semantics can be partitioned into static semantics and execution semantics. Static semantics are implemented by the constraints and type system rules. Execution semantics denote the observable behavior of a program p as it is executed. We look at both aspects in this chapter; but we refer to execution semantics if we don't explicitly say otherwise.

Using a function OB that defines this observable behavior we can define the semantics of a program p_{L_D} by mapping it to a program q in a language for D_{-1} that has the same observable behavior:

There are also a number of approaches for formally defining semantics independent of operational mappings to target languages. However, they don't play an important role in real-world DSL design, so we don't address them in this book.

$$semantics(p_{L_D}) := q_{L_{D-1}} \quad where\ OB(p_{L_D}) == OB(q_{L_{D-1}})$$

Equality of the two observable behaviors can be established with a sufficient number of tests, or with model checking and proof (which is a lot of effort and hence rarely done). This definition of semantics reflects the hierarchy of domains and works both for languages that describe only structure, as well as for those that include behavioral aspects.

The technical implementation of the mapping to D_{-1} can be provided in two different ways: a DSL program can literally be transformed into a program in ~~an~~ L_{D-1}, or an interpreter can be written in L_{D-1} or L_{D_0} to execute the program. Before we spend the rest of this section looking at these two options in detail, we first briefly look at static semantics.

2.3.1 Static Semantics/Validation

Before establishing the execution semantics by transforming or interpreting the program, its static semantics has to be validated. Constraints and type systems are used to this end and we describe their implementation in Part III of the book. Here is a short overview.

■ *Constraints* are simply Boolean expressions that check some property of a model. For example, one might verify that the names of a set of attributes of some entity are unique. For a model to be statically correct, all constraints have to evaluate to `true`. Constraint checking should only be performed for a model that is structurally/syntactically correct[18].

[18] In projectional systems, you cannot build structurally/syntactically incorrect programs in the first place. For parser-based systems, the AST, on which the constraint checks are performed, often is not constructed unless the syntactic structure is correct. This automatically leads to constraint checks being performed only on structurally/syntactically correct models.

Sometimes constraints are used instead of grammar rules. For example, instead of using a $1..n$ multiplicity in the grammar, I often use $0..n$ together with a con-

Embedded C: One driver in selecting the linguistic abstractions that go into a DSL is the ability to easily implement meaningful constraints. For example, in the state machine extension it is trivial to find states that have no outgoing transitions (dead end, Fig. 2.4). In a functional language, such a constraint could be written as `states.select(s|!s.isInstanceOf(StopState)).select(s|s.transitions.size == 0)`. ◀

When defining languages and transformations, developers often have certain constraints in their mind which they consider obvious. They assume that no one would ever use the language in a particular way. However, DSL users may be creative and actually use the language in that way, leading the transformation to crash or create non-compilable code. Make sure that all constraints are actually implemented. This can sometimes be hard. Only extensive (automated) testing can prevent these problems from occurring.

In many cases, a multi-stage transformations is used where a model expressed in L_3 is transformed into a model expressed in L_2, which is then in turn transformed into a program expressed in L_1[19]. Make sure that *every* valid program in L_3 leads to a valid program in L_2. If the processing of L_2 fails with an error message using abstractions from L_2 (e.g., compiler errors), users of L_3 will not be able to understand them; they may have never seen the programs generated in L_2. Again, automated testing is the way to address this issue.

[19] Note how this also applies to the classical case where L_2 is your DSL and L_1 is a GPL which is then compiled!

If many or complex constraints (or type system rules) are executed on a large model, performance may become an issue. Even if the DSL tool is clever about this and only revalidates the constraints for those program elements that changed, it can still be a problem if some kind of whole-model validation is tied to a particular element. To solve this problem, many DSL tools allow users to classify the constraints according to their cost (i.e. performance overhead). Cheap constraints are executed for each changing program element, in real-time, as it changes. Progressively more expensive constraints are checked, for example, as a fragment is saved or only upon explicit request by the user.

■ *Type Systems* are a special kind of constraints. Consider the example of `var int x = 2 * someFunction(sqrt(2));`. The type system constraint may check that the type of the variable is the same or a supertype of the type of the initialization expression. However, establishing the type of the init expression is non-trivial, since it can be an arbitrarily complex expression. A type system is a formalism or framework for defining the rules to establish the types of arbitrary expressions, as well as type checking constraints. It is a form of constraint checking. We cover the implementation of type systems in Part

III of the book (Section ??).

When designing constraints and type system in a language, a decision has to be made between one of two approaches: (a) declaration of intent and checking for conformance and (b) deriving characteristics and checking for consistency. Consider the following examples.

Embedded C: Variables have to be defined in the way shown above, where a type has to be specified explicitly. A type specification expresses the intent that this variable be, for example, of type `int`. Alternatively, a type system could be built to automatically derive the type of the variable declaration based on the type of the `init` expression, an approach called type inference. This would allow the following code to be written: `var x = 2 * someFunction(sqrt(2));`. Since no type is explicitly specified, the type system will infer the type of `x` to be the type calculated for the `init` expression. ◀

Embedded C: State machines that are supposed to be verified by the model checker have to be marked as *verified*. In that case, additional constraints kick in that report certain ways of writing actions as invalid, because they cannot be handled by the model checker. An alternative approach could check a state machine whether these "unverifiable" ways of writing actions are used, and if so, mark the state machine as not verifiable. ◀

Pension Plans: Pension plans can inherit from other plans (called the base plan). If a pension calculation rule overrides a rule in the base plan, then the overriding rule has to marked as *overrides*. This way, if the rule in the base plan is removed or renamed, validation of the sub plan will report an error. An alternative design would simply infer the fact that a rule overrides another one if they have the same name and signature. ◀

Note how in all three cases the constraint checking is done in two steps. First we declare an *intent* (variable is intended to be `int`, this state machine is intended to be verifiable, a rule is intended to override another one). We then *check* if the program conforms to this intention. The alternative approach would infer the fact from the program (the variable's type is whatever the expression's type evaluates to, state machines are verifiable if the "forbidden" features aren't used, rules override another one if they have the same name and signature) *without* any explicitly specified intent.

When designing constraints and type systems, a decision has to be made regarding when to use which approach. Here are some tradeoffs. The specification/conformance approach requires a bit more code to be written, but results in more meaningful and specific error messages. The message can express that fact that one part of a program does not conform to a specification made by another part of the program[20]. The derivation/consistency approach is less effort to write and can hence be seen to be more convenient, but it requires more effort in constraint checking, and error messages may be harder to understand because of the missing, explicit "hard fact" about the program.

[20] It also reduces the chance that users do something they do not intend by mistake. For example, a user might not *want* to override a method from the base class, but it might happen just because the user uses the same name and parameters.

The specification/conformance approach can also be used to "anchor" the constraint checker, because a fixed fact about the program is explicitly given instead of having it derived from a (possibly large) part of the program. This decouples models and can increase scalability. Consider the following example. A program contains a function call, and the type checker needs to check the typing for this call. To do so, it has to determine the type of the called function. Assume this function does *not* specify the return type explicitly, instead it is inferred from the returned expressions. These expressions may be calls to other functions, so the process repeats. In the worst case, a whole chain of function calls must be tracked down in this way to calculate the type of the function initially called by your program. Notice that this requires accessing all the downstream programs, so these all have to be loaded and type checked! In large systems, this can lead to serious performance and scalability issues[21]. If, instead, the type of the called function were given explicitly, no downstream models need to be accessed or loaded.

[21] We have seen such problems in practice with large-scale models.

■ *Multi-Level Constraints* Several sets of constraints can be used to enforce multiple levels of correctness/strictness/compliance for models. The first level are typically basic constraints (such as name uniqueness) and typing rules. They are checked for every program. Additional levels are often optional. They are triggered either by a configuration switch or by using the programs for a given purpose. Additional levels always constrain programs *further* relative to more basic levels.

Embedded C: A nice example of multi-level constraints can be seen in the state machines extension to C. Structural and type system correctness (for C and for state machines) is always checked for every program. However, if a state machine is marked as `verifiable`, then the action code is further restricted via additional constraints. For example, it is not allowed to read and write the same variable within a single transition (i.e. in the total of exit actions, transition actions and entry actions). This is necessary to keep the complexity of the generated model checker input code within limits. ◀

Embedded C: Another example concerns the use of floating point types. Some target devices may not have floating point units (FPUs) which means that floating point types (`float`, `double`) cannot be used in programs that should be deployed on such a target device. So, as the user changes the target device in the build configuration, additional constraints are checked that report floating point types as errors if the target has no FPU). ◀

2.3.2 *Establishing the Correctness of the Execution Engine*

Earlier we have defined the meaning of the program p at D_n as the equivalent observable behavior of a program q at D_{n-1}. This essentially *defines* the transformation or interpreter to be correct. However, this is useless in practice. As the language developer, we have a certain behavior in mind, and we want to make sure that the executing DSL program exhibits this behavior. We have to make sure that the execution engine executes the DSL program accordingly.

In classical programming, we write the GPL code based on our understanding of the requirements. We then write unit tests, based on the same understanding, which test the code we wrote before (Fig. 2.5).

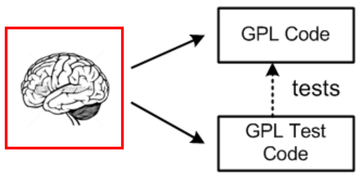


Figure 2.5: Test code tests the application code based on a single understanding of the requirements towards the system.

In DSL testing, we write the DSL, the DSL program and the execution engine based on our understanding of the requirements towards the system. We can still write unit tests (in the GPL) based on this understanding to check for the correctness of the executing DSL program (Fig. 2.6).

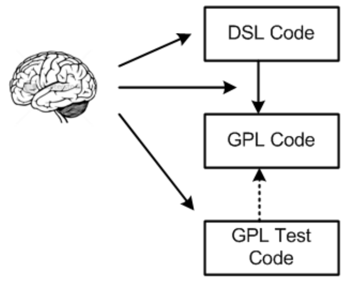


Figure 2.6: Using DSLs, a test written on $D-1$ tests the D program *as well as the transformation* from D to $D-1$.

Writing one DSL program and one unit test ensures that this one program executes correctly with regards to the test case. Our goal here is, however, to ensure that the transformation is correct *for all programs* we can write with the DSL. This can be achieved by writing many DSL programs and many tests – enough to make sure that every branch of the transformation is covered at least once[22]. As always in testing, we encounter the coverage problem: we have to write enough example programs and tests to cover all aspects of the language and the execution engine. In particular, we have to *first think of the corner cases* to write tests for them[23].

A variant of this approach is to express the test cases in the DSL (after extending the DSL with a ways to express tests) and then executing the application code and the test code on the target platform together Fig. 2.7. This is often more convenient since the tests can be formulated more concisely on the level of the DSL. As we will see later, this approach is especially useful if we have *several* execution engines: we write the test once and then execute it on all execution engines.

Note that the GPL program may have *additional, unintended* behaviors not prescribed by the DSL. These can often be exploited mali-

[22] For the Xpand code generator there is a coverage analysis tool that can be used to make sure that for a given test suite, all branches of the transformation template had been executed at least once.

[23] The coverage problem can be solved in some cases by automatic test case generation and formal verification. We discuss this later in this chapter.

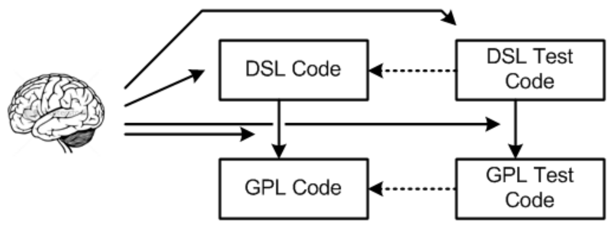


Figure 2.7: Test cases can also be expressed with the DSL and then executed on the target platform together with the application code.

ciously and are known as safety or security problems. These will not be found by testing the GPL code based on the requirements, but only by "trying to exploit" the program (known as penetration testing).

We will elaborate more on ensuring the correctness of the execution semantics in this chapter, as well as in the Part III chapter on DSL testing (Section ??), where we discuss the implementation aspects of DSL and IDE testing.

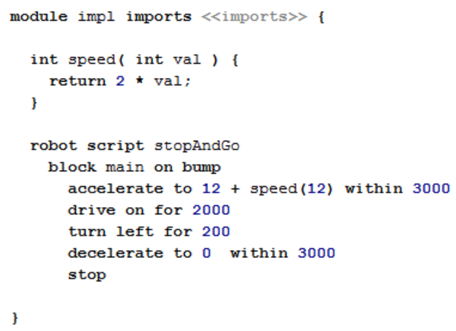
2.3.3 *Transformation*

Transformations define the execution semantics of a DSL by mapping it to another language. In the vast majority of cases a transformation for L_D recreates those patterns and idioms in L_{D-1} for which it provides linguistic abstraction. The result may be transformed further, until a level is reached for which a language with an execution infrastructure exists – often D_0. Code generation, the case where we generate GPL code from a DSL is thus a special case where L_{D_0} code is generated.

Embedded C: The semantics of state machines are defined by their mapping back to C `switch` statements. This is repeated for higher *D* languages. The semantics of the robot control DSL (Fig. 2.8) is defined by its mapping to state machines and tasks (Fig. 2.9). To explain the semantics to the users, prose documentation is available as well. ◄

Component Architecture: The component architecture DSL only described interfaces, components and systems. This is all structure-only. Many constraints about structural integrity are enforced, and a mapping to a distribution middleware is implemented. The formal definition of the semantics are implied by the mapping to the executable code. ◄

DSL programs may be mapped to *multiple* languages at the same time. Typically, there is one primary language that is used for execution of the DSL program (C in Fig. 2.10). The other languages may be used to configure the target platform (generated XML files) or provide input for verification tools (NuSMV in Fig. 2.10). In this case, one



```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump
      accelerate to 12 + speed(12) within 3000
      drive on for 2000
      turn left for 200
      decelerate to 0  within 3000
      stop

}
```

Figure 2.8: The robot control DSL is embedded in C program modules and provides linguistic abstractions for controlling a small lego car. It can accelerate, decelerate and turn left and right.

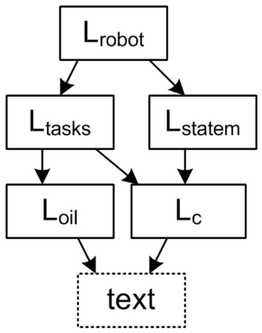


Figure 2.9: Robot control programs are mapped to state machines and tasks. State machines are mapped to C, and tasks are mapped to C as well as to operating system configuration files (a so-called OIL file). In the end, everything ends up in text for downstream processing by existing tools.

has to make sure that the semantics of all generated representations is actually the same. We discuss this problem in Section 2.3.7.

Embedded C: The state machines can be transformed to a representation in NuSMV, which is a model checker that can be used to establish properties of state machines by exhaustive search. Examples properties include freedom from deadlocks, assuring liveness and specific safety properties such as "it will never happen that the out events *pedestrian light green* and *car light green* are set at the same time". ◀

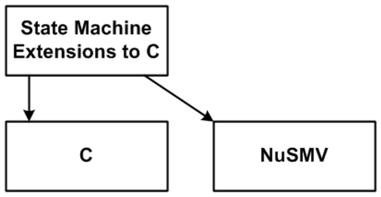


Figure 2.10: From the C state machine extensions, we generate low-level C code (for execution) as well an input file for the NuSMV model checker (for verification).

■ *Multi-staged Transformation* There are several reasons why the gap between a language at D and its target platform may not be bridged by a single transformation. Instead, the overall transformation becomes a chain of subsequent transformations, an approach also known as cascading.

Multi-staged transformation is a form of modularization, and so the reason for doing it the same reason we always use for modularization: breaking down a big problem into a set of smaller problems that can be independently solved. In the case of transformations, this "big problem" is a big semantic gap between the DSL and the target language.[24]. Modularization breaks down this big semantic gap into several smaller ones, making each of them easier to understand, test and maintain[25].

[24] One could measure this semantic gap between two languages: how many constructs do two languages share, how many "synonyms" exist, how many constructs are the same but have different meanings? In practice, the size of the gap is an intuition of the transformation designer.

[25] This approach can only be used if the tools support multi-staged transformation well. This is not true for all DSL tools.
This is on of the reasons why we usually generate GPL source code from DSLs, and not machine code or byte code: we want to reuse existing transformations and optimizations provided by the GPL compiler.

Another reason for multi-stage transformations is the potential for reuse of each of the stages (Fig. 2.11). Reusing lower D languages and their subsequent transformations also implies reuse of potentially non-trivial analyses or optimizations that can be done at that particular abstraction level. Consider GPL compilers as an example. They can be retargetted relatively easily by exchanging the backends (machine code generation phases) or the frontend (programming language parsers and analyzers). For example, GCC can generate code for many different processor architectures (exchangeable backends), and it can generate backend code for several programming languages, among them C, C++ and Ada (exchangeable frontends). The same is possible for DSLs. The same high D models can be executed differently by exchanging the lower D intermediate languages and transformations. Or the same lower D languages and transformations can be used for different higher D languages, by mapping these different languages to the same intermediate language.

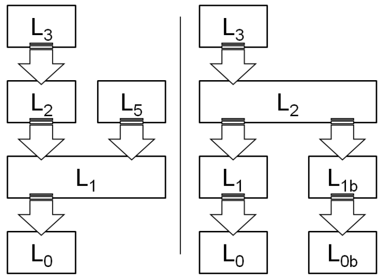


Figure 2.11: *Left:* Backend-Reuse. Different languages (L_3/L_2 and L_5) are transformed to the same intermediate language L_3, reusing its backend generator to L_0. *Right:* Frontend reuse. L_3 is mapped to L_2, which has wto sets of backend generators, reusing the L_3 to L_2 transformation.

Embedded C: The embedded C language (and some of its higher D extensions) have various translation options, for several different target platforms (Win32 and Osek), an example of backend reuse. All of them are C code, but we generate different idioms in the code and different make files. ◀

Multi-stage transformation can also be a natural consequence of incremental language extension along the domain hierarchy, where we repeatedly build additional higher level languages on top of lower level languages. When transforming the higher level languages, it is natural and obvious to transform them onto the next lower level, and not onto a language at D_0.

Embedded C: The extensions to C are all transformed back to C idioms during transformation. Higher-level DSLs, for example, a simple DSL for robot control, are reduced to C plus some extensions such as state machines and tasks (Fig. 2.12), reusing the transformations for those abstractions back to C. ◀

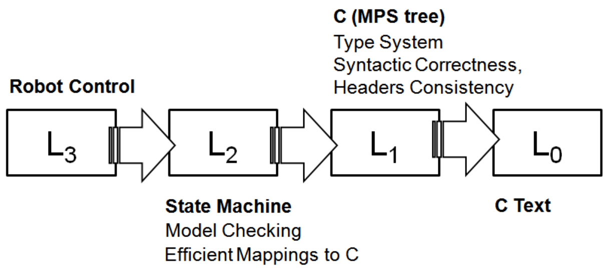


Figure 2.12: Multi-Stage transformation in mbeddr. MPS supports multi-stage transformations really well, so managing the interplay of the set of transformations is feasible.

A special case of a multi-staged transformation is a preprocessor to a code generator. Here, a transformation reduces the set of used language concepts in a fragment to a minimal core, and only the minimal core is supported in the code generator. Note how, in this case, the source and target languages of the transformation are the same. However, the target model only uses a *subset* of the concepts defined by the source/target language. A preprocessor simplifies portability of the actual code generator: it becomes simpler, since only the subset of the language has to be mapped to code.

Embedded C: Consider the case of a state machine where you want to be able to add an "emergency stop" feature, i.e. a new transition from each existing state to a new STOP state. Instead of handling this case in the code generator a model transformation script preprocesses the state machine model and adds all the new transitions and the new emergency stop state (Fig. 2.13). Once done, the existing generator is run unchanged. You have effectively modularized the emergency stop concern into a preprocessor transformation. ◀

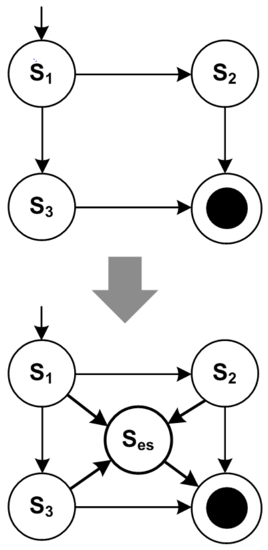


Figure 2.13: A transformation adds an emergency stop feature to a state machine. A new state is added (S_{es}), and a transition from each other state to that new state is added as well. The transition is triggered by the *emergency stop* event (not shown).

Component Architecture: The DSL describes hierarchical component architectures (where components are assembled from interconnected instances of other components). Most component runtime platforms don't support such hierarchical components, so you need to "flatten" the structure for execution. Instead of trying to do this in the code generator, you should consider a model transformation step to do it, and then write a simpler generator that works with a flattened, non-hierarchical model. ◀

Multi-Stage transformations can be challenging. It becomes harder to understand what is going on in total. Debugging the overall transformation can become hard, and good tool support is needed[26]

[26] MPS addresses this problem by (optionally) keeping all intermediate models around for debugging purposes. The language developer can select a program element in any of the intermediate models and have MPS show a trace where the element was transformed from, and where it was transformed to. MPS also shows the transformation code involved in each step.

■ *Efficiency and Optimization* Transforming from D to D_{-1} allows using sophisticated optimizations, potentially resulting in very efficient code. DSL uses domain-specific abstractions and hence includes a lot of domain semantics, so optimizations can take advantage of this and produce very efficient D_{-1} code. However, building such optimizations can be very expensive. It is especially hard to build *global* optimizations that require knowledge about the structure or semantics of large or diverse parts of the overall program. Also, an optimization will always rely on some set of rules that determine when and how to optimize. There will always be corner cases where an experienced developer will be able to write more efficient D_{-1} code manually. However, this requires a competent developer and, usually, a lot of effort *for each specific program*. A tool (i.e. the transformation in this case) will typically address the 90% case well: it will produce reasonably efficient code in the vast majority of cases with very little effort (once the optimizations have been defined). In most cases, this is good enough – in the remaining corner cases, D_{-1} has to be written manually[27].

[27] This argument pro tools is used in GPLs for garbage collection and optimizing compilers for higher level programming languages.

■ *Care about Generated Code* Ideally, generated code is a throw-away artifact, like object files in a C compiler. However, that's not quite true. At least during development and test of the generator you may have to read, understand and debug the generated code. For incomplete DSLs[28], i.e. those where parts of the resulting program have to be written manually in L_{D-1}, readability and good structure is even more important, because the manually written code has to be integrated with the generated parts of the L_{D-1} program. Hence, generated code should use meaningful abstractions, should be designed well, use good names for identifiers, be documented well, and be indented correctly. In short, generated code should generally adhere to the same standards as manually written code. This also helps to diffuse some of the skepticism against code generation that is still widespread in some organizations. However, there are several exceptions to this rule:

[28] We cover completeness in Section 2.5.

Note that in complete languages (where 100% of the L_{D-1} code is generated), the generated code is never seen by a DSL user. But even in this case, concerns for code quality apply and the code has to be understood and tested during DSL and generator development.

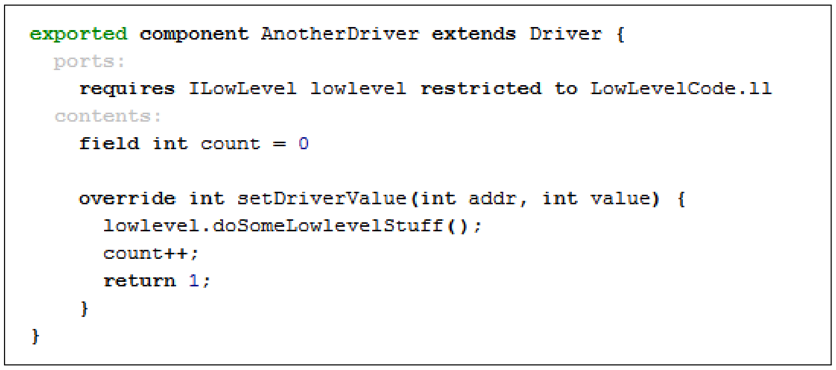
- Sometimes generating really well structured code makes the generator *much* more complicated. You then have to decide whether you want to live with some less nicely structured generated code, or whether you want to increase generator complexity – a valid trade-off, since the generator also needs to be maintained! A good example is *import* statements when generating Java code. It can be a lot of work to find out exactly which imports are needed in a generated class. In this case it may be better to keep the generator simple and use fully qualified class names throughout the code, and/or to import a few too many classes[29].

[29] Xtend provides special support for this problem based on an `ImportManager`. It makes generating the correct imports relatively simple.

- Using a generator opens up additional options you wouldn't consider when writing code manually (and which are hence considered ugly). An example is generated collection classes. Imagine your models define entities, and from each entity you generate a Java Bean. In Java version 1.4 and earlier, Java did not have generics, so in order to work with collections of entities you would use the generic `List` class. In the context of generated code you might want to consider generating a specific collection class for each entity, with an API typed to the respective Java Bean. This makes live much more convenient for those people who write Java code that *uses* the generated Beans.

- The third exception to the rule is if the code has to be highly optimized for reasons of performance and code size. While you can still indent your code well and use meaningful names, the *structure* of the code may be convoluted. Note, however, that the code would look the same way if it were written by hand in that case.

  Embedded C: The components extension to C supports components with provided and required ports. A required port declares which interface it is expected to be connected to. The same interface can be provided by different components, implementing the interface differently. Upon translation of the component extension, regular C functions are generated. An outgoing call on a required port has to be routed to the function that has been generated to implement the called interface operation in the target component. Since each component can be instantiated multiple times, and each instance can have its required ports connected to *different* component instances (implementing the same interface) there is no way for the generated code to know the particular function that has to be called for an outgoing call on a required port for a given instance. An indirection through function pointers in used instead. Consequently, functions implementing operations in components take an additional `struct` as an argument which provides those function pointers for each operation of each required port. A call on a required port thus is a relatively ugly affair based on function pointers. However, to achieve the desired goal, no different, cleaner code approach is possible in C. It is optionally possible to *restrict* a required port to a particular component (Fig. 2.14). In this case, the target function is known statically and no function pointer-based indirection is required. The resulting code is cleaner and more efficient. Programmers trade flexibility for performance. ◄



```
exported component AnotherDriver extends Driver {
  ports:
    requires ILowLevel lowlevel restricted to LowLevelCode.ll
  contents:
    field int count = 0

    override int setDriverValue(int addr, int value) {
      lowlevel.doSomeLowlevelStuff();
      count++;
      return 1;
    }
}
```

Figure 2.14: The required port `lowlevel` is not just bound to the `ILowLevel` interface, but restricted tot the `ll` port of the `LowLevelCode` component. This way, it is statically known which C function implements the behavior and the generated code can be optimized.

■ *Platform* Code generators can become complex. The complexity can be reduced by splitting the overall transformation into several steps – see above. Another approach is to work with a manually implemented, rich domain specific platform. It typically consists of middleware, frameworks, drivers, libraries and utilities that are taken advantage of by the generated code.

Where the generated code and the platform "meet" depends on the complexity of the generator, requirements regarding code size and performance, the expressiveness of the target language and the potential availability of libraries and frameworks that can be used for the task. In the extreme case, the generator just generates code to populate/configure the frameworks (which might already exist, or which you have to grow together with the generator) or provides statically typed facades around otherwise dynamic data structures. Don't go too far towards this end, however: in cases where you need to consider resource or timing constraints, or when the target platform is predetermined and perhaps limited, code generation is the better approach: trying to make the platform too generic or flexible will increase *its* complexity.

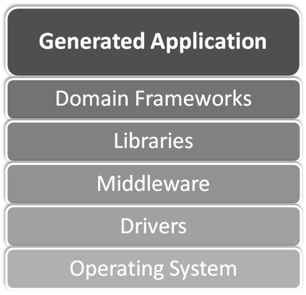


Figure 2.15: Typical layering structure of an application created using DSLs.

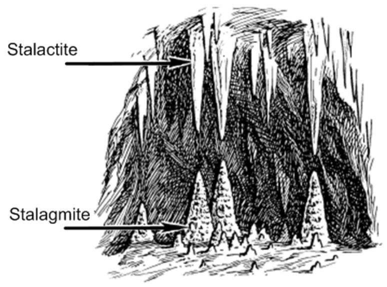


Figure 2.16: Stalagmites and stalactites in limestone caves as a metaphor for a generator and a platform: the stalagmite represents the platform, it grows from up the lower abstraction levels. Stalagtites represent the transformations, which grow down from the high abstraction level represented by the DSL.

Embedded C: For most aspects, we use only a very shallow platform. This is mostly for performance reasons and for the fact that the subset of C that is often used for embedded systems does not provide good means of abstraction. For example, state machines are translated to `switch` statements. If we were to generate Java code in an enterprise system, we may populate a state machine framework instead. In contrast, when we translate the component definitions to the AUTOSAR target environment, a relatively powerful platform is used – namely the AUTOSAR APIs, conventions and generators. ◀

2.3.4 *Interpretation*

An interpreter is basically a program that acts on the DSL program it receives as an input. How it does that depends on the particular paradigm used (see Section 3.2). For imperative programs it steps through the statements and executes their side effects. In functional programs, the interpreter (recursively) evaluates functions. For declarative programs, some other evaluation strategy, for example based on a solver, may be used. We describe some of the details about how to design and implement interpreters in Section **??**.

Refrigerators: The DSL also supports the definition of unit tests for the asynchronous, reactive cooling algorithm. These tests are executed with an in-IDE interpreter. A simulation environment allows the interpreter to be used interactively. Users can "play" with a cooling program, stepping through it in single steps, watching values change. ◀

Pension Plans: The pension DSL supports the in-IDE execution of rule unit tests by an interpreter. In addition, the rules can be debugged. The rule language is functional, so the debugger "expands" the calculation tree, and users can inspect all intermediate results. ◀

For interpretation, the domain hierarchy could be exploited as well: the interpreter for L_D could be implemented in L_{D-1}. However, in practice we see interpreters written in L_{D_0}. They may be extensible, so new interpreter code can be added in case specialized languages define new language concepts.

The abstraction level of an interpreter must be decided. One alternative might ignore for example the use of registers when performing an assignment, avoiding problems resulting from parallelism. Alternatively, the interpreter might model everything and thereby can address issues related to parallelism. In other words, an interpreter defines a virtual machine^ and it is fundamental that this virtual machine has an adequate abstraction level and/or the users are aware of exactly what it means for the execution of the program on the target hardware if the program runs on the virtual machine.

2.3.5 *Transformation vs. Interpretation*

When defining the execution semantics for a language, a decision has to be made between transformation (code generation) and interpretation. Here are a couple of criteria to help with this decision.

Code Inspection When using code generation, the resulting code can be inspected to check whether it resembles code that had previously been written manually in the DSL's domain. Writing the transformation rules can be guided by the established patterns and idioms in L_{D-1}. Interpreters are meta programs and as such harder to relate to existing code patterns.

Debugging Debugging generated code is straight forward if the code is well structured (which is up to the transformation) and an execution paradigm is used for which a decent debugging approach exists (not the case for many declarative approaches). Debugging interpreters is harder, because, they are meta programs. For example, setting breakpoints in the DSL program requires the use of conditional breakpoints in the interpreter, which are typically cumbersome to use[30]

[30] This is especially useful during the development of the execution engine. Once a the DSL and the engine are finished, users should be able to debug DSL programs on the level of the DSL. However, since building DSL debuggers is not directly supported by most language workbenches, this is a lot of work – and users are required to debug on L_{D-1}.

Performance and Optimization The code generator can perform optimizations that result in small and tight generated code. The compiler for the generated code may come with its own optimizations which are used automatically if source code is generated and subsequently compiled, simplifying the code generator[31]. Generally, performance is better in generated environments, since interpreters always imply an additional layer of indirection during the execution of the program.

[31] For example, it is not necessary to optimize away calls to empty functions, **`if`** statements that always evaluate to **`true`**, or arithmetic expressions containing only constants.

Platform Conformance Generated code can be tailored to any target platform. The code can look exactly as manually written code would look, no support libraries are required. This is important for systems where the source code (and not the DSL code) is the basis for a contractual obligations or for review and/or certification. Also if artifacts need to be supplied to the platform that are not directly executable (descriptors, meta data), code generation is more suitable.

Modularization When incrementally building DSLs on top of existing languages, it is natural to use transformations to L_{D-1}. [32]

[32] While it is theoretically possible to also extend interpreters incrementally along a hierarchy of languages, I have not seen this in practice. Interpreters are typically written in a GPL.

Turnaround Time Turnaround time for interpretation is better than for generation: no generation, compilation and packaging step is required. Especially for target languages with slow compilers, large amounts of generated code can be a problem.

Runtime Change In interpreted environments, the DSL program can be changed as the target system runs; the DSL editor can even be integrated into the target system[33].

[33] The term data-driven system is often used in this case.

Refrigerators: There were two reasons for implementing the interpreter for the cooling programs. The first was that initially we didn't have a code generator because the target architecture was not yet defined. To be able to execute cooling programs, we needed an interpreter and simulator. Second, the turn-around time for the domain experts as the experimented with the DSL programs is much reduced compared to generating, compiling and running C code. The (interpreted) simulator also allowed the domain experts to run the programs at a speed they could follow. This proved an important means of understanding and debugging the asynchronous reactive cooling programs. ◀

Embedded C: This DSL exploits incremental extension to the C programming language (inductive DSL definition). In this case it is natural to use transformation to L_{D-1} as a means of defining the semantics of extensions. Also, since the target domain is embedded software, performance, code size and reuse of the optimizations provided by the C compiler is essential. Interpretation was never an option. ◀

Component Architecture: The driving factor for using generation over interpretation was platform conformance. The reason for the DSL is to automate the generation of target platform artifacts and thereby make working with the platform more efficient. ◀

Pension Plans: Turnaround time was important for the pension contract specification. Also, the domain experts, as the created the pension plans, did not have access to the final execution platform. An in-IDE interpreter was clearly the best choice. ◀

WebDSL: Platform conformance was key here. Web applications have to use the established web standards, and the necessary artifacts have to be generated. An interpreted approach would not work in this scenario. ◀

Combinations between the two approaches are also possible. For example, transformation can create an intermediate representation which is then interpreted. Or an interpreter can generate code on the fly as a means of optimization. While this approach is common in GPLs (e.g., the JVM), we have not seen this approach used for DSLs.

2.3.6 *Sufficiency*

A program fragment is *sufficient for transformation T* if the fragment itself contains all the data necessary to executed for the transformation. While dependent fragments are by definition not sufficient without the transitive closure of fragments they depend on, an independent fragment may be sufficient for one transformation, and insufficient for another.

Refrigerators: The hardware structure is sufficient for a transformation that generates an HTML doc that describes the hardware. It is insufficient regarding the C code generator, since the behavior fragment is required as well. ◀

Sufficiency is important when large systems are concerned. An sufficient fragment can be used for code generation without checking out and/or loading other fragments. This supports modular, incremental transformations of only the changed fragments, and hence, potentially significant improvements in performance and scalability.

2.3.7 *Synchronizing Multiple Mappings*

Ensuring the semantics of the execution engine becomes more challenging if we transform the program to *several different* targets using several different transformations. We have to ensure that the semantics of all resulting programs are identical[34]. In practice, this case often occurs if an interpreter is used in the IDE for "experimenting" with the models, and a code generator creates efficient code for execution in the target environment. To synchronize the semantics in this case we recommend providing a set of test cases that are expressed on DSL level, and that are executed in all executable representations, expecting them to succeed in all. If the coverage of these test cases is high enough to cover all of the observable behavior, then it can be assumed with reasonable certainty that the semantics are indeed the same[35].

[34] At least to the extent we care – we may not care if one of the resulting programs is faster or more scalable. In fact, these differences may be the very reason for having several mappings)

[35] Strictly speaking they are just bug-compatible, i.e. they may all make the same mistakes.

Pension Plans: The unit tests in the pension plans DSL are executed by an interpreter in the IDE. However, as Java code is generated from the pension plan specifications, the same unit tests are also executed by the generated Java code, expecting the same results as in the interpreted version. ◀

Refrigerators: A similar situation occurs with the cooling DSL where an in IDE-interpreter is used for testing and experimenting with the models, and a code generator creates the executable version of the cooling algorithm that actually runs on the microcontroller in the refrigerator. A suite of test cases is used to ensure the same semantics. ◀

2.3.8 *Choosing between Several Mappings*

Sometimes there are several *alternative* ways how a program in L_D can be translated to a single L_{D-1}, for example to realize different non-functional requirements (optimizations, target platform, tracing or logging). There are several ways how one alternative may be selected.

- In analogy to compiler switches, the decision can be controlled by additional external data. Simple parameters passed to the transformation are the simplest case. A more elaborate approach is to have an additional model, called an annotation model, which contains data used by the transformation to decide how to translate the core program. The transformation uses the L_D program and the annotation model as its input. There can be several different annotation models for the same core model that define several different transformations, to be used alternatively. An annotation model is a separate viewpoint (Section 2.4) an can hence be provided by a different stakeholder than the one who maintains the core L_D program.

- Alternatively, L_D can be extended to directly contain additional data to guide the decision. Since the data controlling the transformation is embedded in the core program, this is only useful if the DSL user can actually decide which alternative to choose, and if only one alternative should be chosen for each program. Annotation models provide more flexibility.

- Heuristics, based on patterns, idioms and statistics extracted from the L_D program, can be used to determine the applicable transformation as well. Codifying these rules and heuristics can be hard though, so this approach is rarely used.

As we have suggested above in the case of multiple transformations of the *same* L_D program, here too extensive testing must be used to make sure that all translations exhibit the same semantics (except for the non-functional characteristics that may be expected to be different, since they often are the reason for the different transformations in the first place).

2.3.9 *Reduced Expressiveness and Verification*

It may be beneficial to limit the expressiveness of a language. Limited expressiveness often results in more sophisticated analyzability. For example, while state machines are not very expressive (compared to fully fledged C), sophisticated formal verification algorithms are available (e.g., model checking using SPIN[36] or NuSMV[37]). The same is true for first-order logic, where satisfiability (SAT) solvers[38] can be used to check programs for consistency. If these kinds of analyses are useful for the model purpose, then limiting the expressiveness to the respective formalism may be a good idea, even if it makes expressing certain programs in D more cumbersome[39]. Possibly a DSL should be partitioned into several sub-DSLs, where some of them are verifiable and some are not.

[36] `http://spinroot.com`

[37] `http://nusmv.fbk.eu/`

[38]

[39] A simple example is to use integers with ranges `int[0..10] x;` instead of general integers. This makes programs harder to write (ranges must be specified every time) but easier to analyze.

Embedded C: This is the approach used here: model checking is provided for the state machines. No model checking is available for general purpose C, so behavior that should be verifiable must be isolated into a state machine explicitly. State machines interact with their surrounding C program in a limited an well-defined way to isolate them and make them checkable. Also, state machines marked as `verifiable` cannot use arbitrary C code in its actions. Instead, an action can only change the values of variables local to the state machine and set output events (which are then mapped to external functions or component runnables). The key here is that the state machine is completely self-contained regarding verification: adapting the state machine to its surrounding C program is a separate concern and irrelevant to the model checker. ◀

However, the language may have to be reduced to the point where domain experts are not able to use the language because the connection to the domain is too loose. To remedy this problem, a language with limited expressiveness can be used at D_{-1}. For analysis and verification, the L_D programs are transformed down to the verifiable L_{D-1} language. Verification is performed on L_{D-1}, mapping the results back to L_D. Transforming to a verifiable formalism also works if the formalism is not at D_{-1}, as long as a mapping exists. The problem with this approach is the interpretation of analysis results in the context of the DSL. Domain users may not be able to interpret the results of model checkers or solvers, so they have to be translated back to the DSL. Depending on the semantic gap between the generated model checker input program and the DSL, this can be very hard.

2.3.10 Documentation

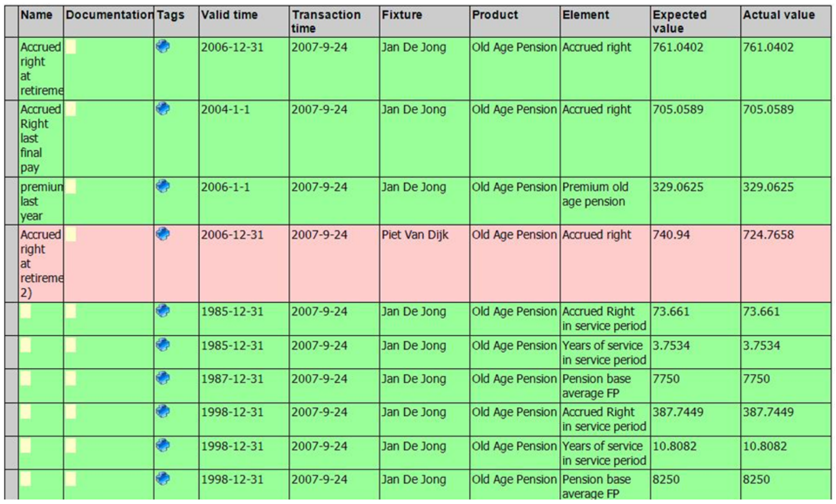
Formally, defining semantics happens by mapping the DSL concepts to D_{-1} concepts for which the semantics is known. For DSLs used by developers, and for domains that are defined inductively (bottom-up), this works well. For application domain DSLs, and for domains defined deductively (top-down), this approach is not necessarily good enough, since the D_{-1} concepts has no inherent meaning to the users and/or the domain. An additional way of defining the meaning of the DSL is required. Useful approaches include prose documentation[40] as well as test cases or simulators. This way, domain users can "play" with the DSL and write down their expectations formally in test cases.

[40] We suggest to always write such documentation in tutorial style, or as FAQs. Hardly anyone reads "reference documentation": while it may be complete and correct, it is boring to read and does not guide users through using the DSL.

Embedded C: The extensible C language comes with a 100 page PDF that shows how to use the MPS-based IDE, illustrates the changes to regular C, provides examples for all C extensions and also discusses how to use the integrated analysis tools. ◀

Refrigerators: This DSL has a separate viewpoint for defining test cases where domain experts can codify their expectations regarding the behavior of cooling programs. An interpreter is available to simulate the programs, observe their progress and stimulate them to see how they react. ◀

Pension Plans: This DSL supports an Excel-like tabular notation for expressing test cases for pension calculation rules (Fig. 2.17). The calculations are functional, and the calculation tree can be extended as a way of debugging the rules. ◀



Name	Documentation	Tags	Valid time	Transaction time	Fixture	Product	Element	Expected value	Actual value
Accrued right at retireme			2006-12-31	2007-9-24	Jan De Jong	Old Age Pension	Accrued right	761.0402	761.0402
Accrued Right last final pay			2004-1-1	2007-9-24	Jan De Jong	Old Age Pension	Accrued right	705.0589	705.0589
premiun last year			2006-1-1	2007-9-24	Jan De Jong	Old Age Pension	Premium old age pension	329.0625	329.0625
Accrued right at retireme 2)			2006-12-31	2007-9-24	Piet Van Dijk	Old Age Pension	Accrued right	740.94	724.7658
			1985-12-31	2007-9-24	Jan De Jong	Old Age Pension	Accrued Right in service period	73.661	73.661
			1985-12-31	2007-9-24	Jan De Jong	Old Age Pension	Years of service in service period	3.7534	3.7534
			1987-12-31	2007-9-24	Jan De Jong	Old Age Pension	Pension base average FP	7750	7750
			1998-12-31	2007-9-24	Jan De Jong	Old Age Pension	Accrued Right in service period	387.7449	387.7449
			1998-12-31	2007-9-24	Jan De Jong	Old Age Pension	Years of service in service period	10.8082	10.8082
			1998-12-31	2007-9-24	Jan De Jong	Old Age Pension	Pension base average FP	8250	8250

Figure 2.17: Test cases in the pension language allow users to specify test data for each input value of a rule. The rules are then evaluated by an interpreter, providing immediate feedback about incorrect rules.

2.4 *Separation of Concerns*

A domain D is may be composed from different concerns. Each concern covers a different aspect of the overall domain. When developing a system in a domain, all the concerns in that domain have to be addressed. Separation into concerns is often driven by different aspects of the system being specified by different stakeholders or at different times in the development process. Fig. 2.18 shows $D_{1.1}$ composed from the concerns A, B and C.

For embedded software, these could be component and interface definitions (A), component instantiation and connections (B), as well as scheduling and bus allocation (C).

Two fundamentally different approaches are possible to deal with the set of concerns in a domain. Either a single, integrated language can be designed that addresses all concerns of D in one integrated model. Alternatively, separate concern-specific DSLs can be defined, each addressing one or more of the domain's concerns[41]. A complete program then consists of a set of dependent, concern-specific frag-

[41] Strictly speaking, this is not quite true: some concerns are typically also addressed by the execution engine. We discuss this below in the section on Cross-Cutting Concerns.

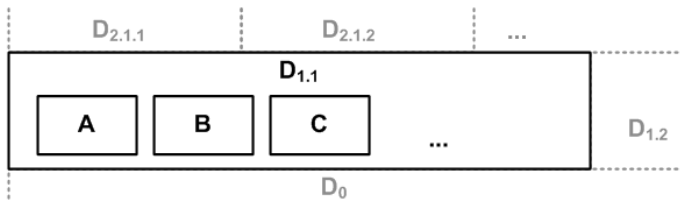


Figure 2.18: A domain may consist of several concerns. A domain is covered either by a DSL that addresses all of these concerns, or by a set of related, concern-specific DSLs.

ments that relate to each other in a well-defined way. Viewpoints support this separation of domain concerns into separate DSL. Fig. 2.19 illustrates the two different approaches.

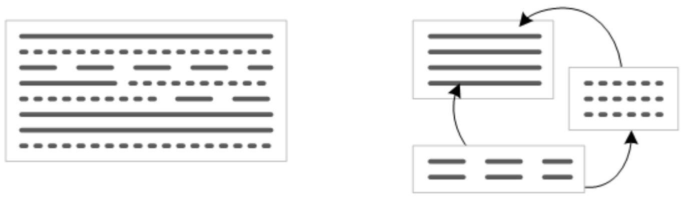


Figure 2.19: *Left:* An integrated DSL, where the various concerns (represented by different line styles) are covered by a single integrated language (and consequently, one model). *Right:* Several viewpoint languages (and model fragments), each covering a single concern. Arrows in Part B highlight dependencies between the viewpoints.

Embedded C: The tasks language module includes the task implementation as well as task scheduling in one language construct. Scheduling and implementation are two concerns that could have been separated. We opted against this, because both concerns are specified by the same person. The language used for implementation code is `med.core`, whereas the task constructs are defined in the `med.tasks` language. So the languages are modularized, but they are used together in a single heterogeneous fragment. ◀

WebDSL: Web programs consists of multiple concerns including persistent data, user interface, and access control. WebDSL provides specific languages for these concerns, but *linguistically integrates* them into a single language[42]. Declarations in the languages can be combined in WebDSL modules. A WebDSL developer can choose how to factor declarations into modules; e.g., all access control rules in one module, or all aspects of some feature together in one module. ◀

[42]

Component Architecture: The specification of interfaces and components is done with one DSL in one viewpoint. A separate viewpoint is used to describe component instantiation and connection. This choice has been made because the same set of interfaces and components will be instantiated and connected *differently* in different usage scenarios, so separate fragments are useful. ◀

2.4.1 Viewpoints for Concern Separation

If viewpoints are used, the concern-specific DSLs, and consequently the viewpoint models, should have well-defined dependencies; cycles should be avoided. If dependencies between viewpoint fragments are kept cycle-free, the independent fragments may be sufficient for certain transformations; this can be a driver for using viewpoints in the first place.

The IDE should provide navigational support: If an element in viewpoint B points to an element in viewpoint A then it should be possible to follow this reference ("Ctrl-Click"). It should also be possible to query the dependencies in the opposite direction ("find the persistence mapping for this entity" or "find all UI forms that access this entity").

The dependent viewpoint fragment (and the language to express it) have to provide a way of pointing to the referenced element. In practice, this usually means that the referenced element has to provide a qualified name that can be used in the reference[43].

[43] In projectional editors one can *technically* use the UUID of the target element for the reference, but for the user, some kind of qualified name is still necessary.

Separating out a domain concern into a separate viewpoint fragment can be useful for several reasons. If different concerns of a domain are specified by different stakeholders then separate viewpoints make sure that each stakeholder has to deal only with the information they care about. The various fragments can be modified, stored and checked in/out separately, maintaining only referential integrity with referenced fragment[44]. The viewpoint separation has to be aligned with the development process: the order of creation of the fragments must be aligned with the dependency structure.

[44] Projectional editors can use a different approach. They can store the information of all concerns in a single model, but then use different projections to address the needs of different stakeholders. This solves the problem of referential integrity. However, this approach does not support separate store and check in/out.

A final (very pragmatic) reason for using viewpoints is when the tooling used does not support embedding of a reusable language because syntactic composition is not supported.

Another reason for separate viewpoints is a 1:n relationship between the independent and the dependent fragments. If a single core concern may be enhanced by several different additional concerns, then it is crucial to keep the core concern independent of the information in the additional concerns. Viewpoints make this possible.

Refrigerators: One concern in this DSL specifies the logical hardware structure of refrigerators installations. The other one describes the refrigerator cooling algorithm. Both are implemented as separate viewpoints, where the algorithm DSL references the hardware structure DSL. Using this dependency structure, different algorithms can be defined for the same hardware structure. Each of these algorithms resides in its own fragment. While the C code generation requires both behavior and hardware structure fragments, the hardware fragment is sufficient for a transformation that creates a visual representation of the hardware structures (see Fig. 2.20). ◄

2.4.2 Viewpoints as Annotation Models

A special case of viewpoint separation is annotation models (already mentioned in Section 2.3.8). An annotation provides additional, often technical or transformation-controlling data for elements in a core program[45]. This is especially useful in a multi-stage transformation (Section 2.3.3) where additional data may have to be specified for the result

[45] For those who know Eclipse EMF: `genmodels` are annotation models for `ecore` models.

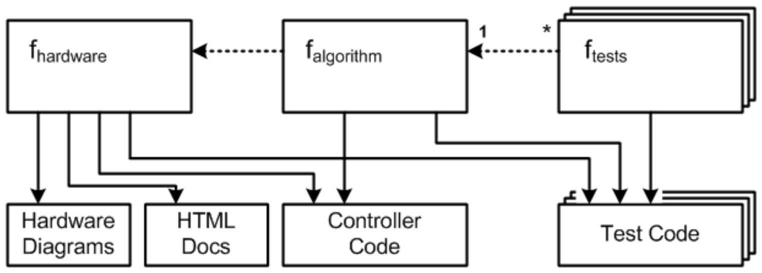


Figure 2.20: The hardware fragment is independent, and sufficient for generation of hardware diagrams and documentation. The algorithms fragment depends on the hardware fragment. The two of them together are sufficient for generating the controller code. Tests depend on the algorithm. There are many test fragments for a single algorithm fragment.

of the first phase to control the execution of the next phase. Since that intermediate model is generated, it is not possible to add these additional specifications to the intermediate model directly. Externalizing it into an annotation model solves that problem.

Example: For example, if you create a relational data model from an object oriented data model, you might automatically derive database table names from the name of the class in the OO model. If you need to "change" some of those names, use an annotation model that specifies an alternate name. The downstream processor knows that the name in the annotation model overrides the name in the original model[46]. ◀

[46] This is a typical example of where the easiest thing in a projectional editor would be to just place a field for holding an overriding table name under program element that represents the table. Users can edit that table name in a special projection.

2.4.3 *Viewpoint Consistency*

If viewpoints are used, constraints have to be defined to check consistency of the viewpoints. A dependent viewpoint fragment contains program elements that reference program elements in another fragment. It is straight forward to check that the target elements of these actually exist, since the reference will break if it does not; in most tools these kinds of checks are available by default.

The other direction is more interesting. Assume two viewpoints: business data structure and persistence mapping. There may be a constraint that says that every `Entity` defined in the business data viewpoint has to have exactly one `EntityPersistenceMapping` element that points to the respective `Entity`. It is an error – incomplete model – if such an `EntityPersistenceMapping` does not exit. Checking this constraint has two problems:

The first problem may be performance. The "whole world" has to be searched to check if a referencing program element exists somewhere. If the tool supports it, this problem can be solved by automatically maintained reverse indices[47].

The second problem is more fundamental: it is not clear what constitutes *the whole world*. The fragment with the persistence mapping for a given `Entity` may reside on a different machine or be under the control of a different user. It may not be accessible to the constraint

[47] This data should also be exploited by the IDE. UI actions should be available to navigate from the referenced element (the `Entity`) to the referencing elements (the `EntityPersistenceMapping`). This is more than generic Find Usages, since it specifically searches for certain kinds of usages (the `EntityPersistenceMapping` in this example). Further tool support may include creation of such referencing elements based on a policy that determines into which fragment the created element should go

checker when the user edits the business data fragment. To solve this problem, it is necessary to define explicitly what *the world* is, using some kind of configuration. For example, a C compiler's include path or Java's classpath are ways of defining the scope within which the overall system description must be complete. This does not necessarily have to be done by each developer who, for example, works on the business data. But at the point when the final system is generated or built, such a "world definition" is essential.

2.4.4 *Cross-Cutting Concerns*

In the discussion so far we have considered concerns that can be modularized clearly. Fig. 2.18 emphasizes this: the concern boxes are neatly arranged next to each other. However, there may also be concerns that do *not* fit into the chosen modularization approach. These are typically called *cross-cutting concerns*, see Fig. 2.21.

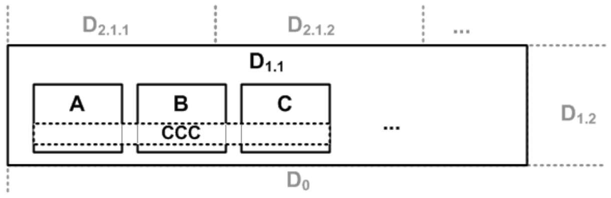


Figure 2.21: Cross-cutting concerns cannot be modularized. The permeate other concerns.

In the context of DSLs we have to separate several classes of cross-cutting concerns:

■ *Handled by Execution Engine* If we are lucky, a concern that is cross-cutting in the domain can be handled completely by the execution engine. For example the collection of performance data, billing information or audit logs typically does not have to be described in the DSL at all. Since every program in the domain has to address this concern in the same way, the implementation can be handled by the execution engine by inserting the respective code at the relevant locations (in case of a generator).

Component Architecture: The component architecture DSL supports the collection of performance data. Using mock objects, we started running load tests early on. For a load test, we have to collect the times it takes to execute operations on components. Based on a configuration switch, the generator would add the necessary code automatically. ◀

■ *Modularized in DSL* Another class of cross-cutting concerns are those that crosscut in the resulting executable system, but can be modularized on DSL level. A good example is permissions. Specifying users, roles and permissions to access ~~certain~~ resources in the system can be modularized into a concern, and it is typically described in a separate viewpoint. It is then the job of the execution engine to consider the specified permissions in all relevant places in the resulting system.

WebDSL: WebDSL has a means to specify access control for web pages. The generator injects the necessary code to check these permissions into the client side and server side parts of the resulting web application. ◀

■ *Cross-Cutting in the DSL* The third class is when the concern ~~cross-cuts~~ the programs written in the DSL and can *not* be modularized, as in the previous class. In this case we have to deal with cross-cutting concerns in the same way as we do it today in programming languages: we either have to manually insert the code in *all* the relevant places in the DSL program, or we have to resort to aspect weaving on DSL level[48].

[48] Building a (typically relatively limited) aspect weaver on DSLs level is not a big problem, since we already have access to the AST, and transforming it in a way where we inject additional code based on a DSL-specific pointcut specification is relatively straight forward.

Component Architecture: We implemented a simple weaver that is able to introduce additional ports into existing components. It was used, among other things, to modularize the monitoring concern: if monitoring was enabled, this aspect component would add the `mon` port to all other components, enabling the `MonitoringConsole` to connect to the other components and query monitoring data (see the code below[49]). ◀

[49] The ∗ specifies that this aspect applies to *all* existing components. Other selectors could be used instead of the ∗ to select only a subset).

```
namespace monitoring feature monitoring {

  component MonitoringConsole ...
  instance monitor: ...
  dynamic connect monitor.devices .. .

  aspect (*) component {
    provides mon: IMonitoring
  }
}
```

2.4.5 *Views on Programs*

In projectional editors it is also possible to store the data for all viewpoints in the same model tree, while using different projections to show different views onto the model to materialize the various viewpoints. The particular benefit of this approach is that additional concern-specific views can be defined later, after programs have been created. It also avoids the need for defining sophisticated ways of referencing program elements from other viewpoints.

MPS also provides so-called annotations, where additional model data can be "attached" to any model element, and shown optionally.

Pension Plans: Pension plans can be shown in a graphical notation highlighting the dependency structure (Fig. 2.22). The dependencies can still be edited in this view, but the actual content of the pension plans is not shown. ◀

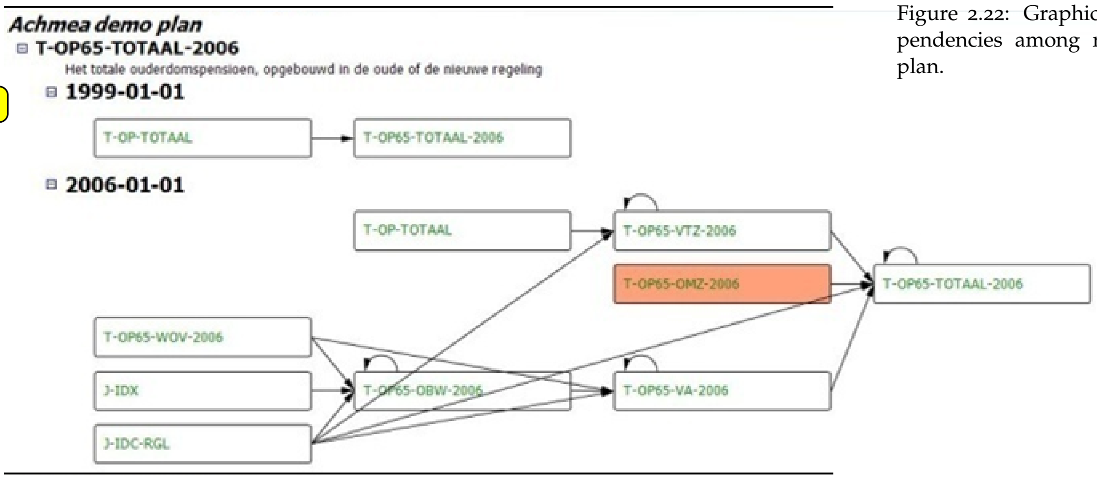
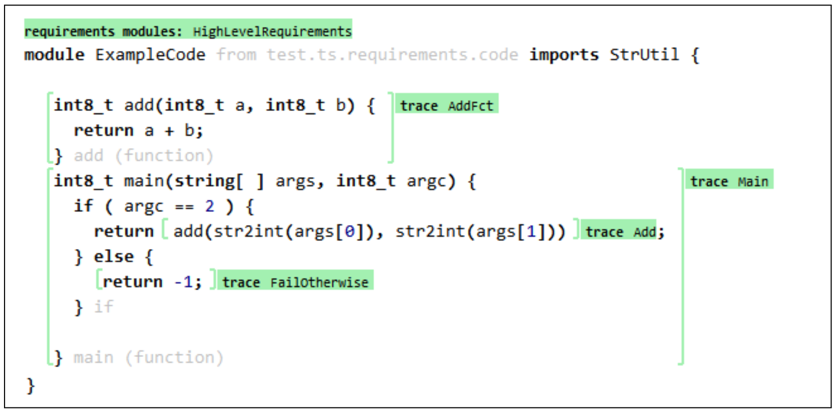


Figure 2.22: Graphical notation for dependencies among rules in a pension plan.

Embedded C: Annotations are used for storing requirements traces and documentation information in the models (Fig. 2.23). The program can be shown and edited with and without requirements traces and documentation text. ◀



```
requirements modules: HighLevelRequirements
module ExampleCode from test.ts.requirements.code imports StrUtil {

  int8_t add(int8_t a, int8_t b) {                 trace AddFct
    return a + b;
  } add (function)
  int8_t main(string[ ] args, int8_t argc) {                   trace Main
    if ( argc == 2 ) {
      return add(str2int(args[0]), str2int(args[1])) trace Add;
    } else {
      return -1; trace FailOtherwise
    } if

  } main (function)
}
```

Figure 2.23: The green annotations are traces into a requirements database. The program can be edited with and without these annotations. The annotations language has no dependency on the languages it annotates.

2.4.6 *Viewpoints for Progressive Refinement*

There is an additional use case for viewpoint models not related to the concerns of a domain, but to progressive refinement. Consider the development of complex systems, which typically proceeds in phases: it starts with requirements, proceeds to high-level component design and specification of non-functional properties, and finishes with the implementation of the components. In each of these phases, models can be used to represent the system with abstractions that are appropriate for the phase. An appropriate DSL is needed to represent the models in each phase (Fig. 2.24). The references between model elements are called traces[50]. Since the same conceptual elements may be represented on different refinement levels (e.g., component design and component implementation), synchronization between the viewpoint models is often required (see next subsection).

[50]

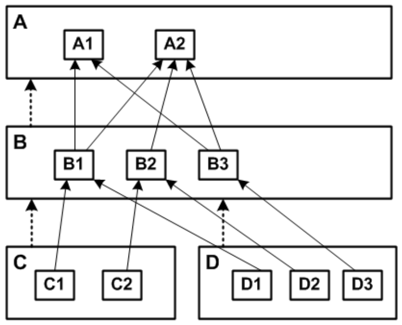


Figure 2.24: Progressive refinement: the boxes represent models expressed with corresponding languages. The dotted arrows express dependencies, whereas the solid arrows represent references between model elements.

2.4.7 *Model Synchronization*

In the discussion of viewpoints so far we have assumed that there is no overlap between the viewpoints: every piece of information lives in exactly on viewpoint. Relationships between viewpoints are established by references (which means that the Referencing language composition technique can be used, as discussed in Section 2.6.1). However, sometimes this is not the case, and the same (conceptual) information is represented in two viewpoint models. Obviously there is a constraint that enforces consistency between the viewpoints; the models have to be synchronized[51].

In some cases the rules for establishing consistency between viewpoints can be described formally, and hence the synchronization can be automated, if the DSL tool supports such synchronization[52]. An example occurs in mbeddr C:

Embedded C: Components implement interfaces. Each component provides an implementation for each method defined in each of the interfaces it implements. If a new method is added to an interface, all components that implement that particular interface must get a new, empty method implementation. This is an example of model synchronization. ◀

In this example the synchronization is trivial, for two reasons: first, there is a clear (unidirectional) dependency between the method implementation and the operation specification in the interface, so the

[51] Thanks to the participants of the MoDELS 2012 workshop on Multi-Paradigm Modeling who, by discussing this issue, reminded me that it is missing from the book.

[52] MPS has a nice way of automatically executing a quick fix for a constraint violation. If the constraint detects an inconsistency between viewpoints, the quick fix can automatically correct the problem. This also solves the *whole world* problem neatly, since every dependent fragment is "corrected" as soon as it is opened in the editor.

synchronization is also unidirectional. Second, the information represented in both models/places is identical, so it is easy to detect an inconsistency and fix it. However, there are several more complicated cases:

- The dependency might be bidirectional, and changes may be allowed in either model. This means that two transformations have to be written, one for each direction, or a formalism for expressing the transformation has to be used that can be executed in both directions[53]. In multi-user scenarios it is also possible that the two models are changed at the same time, in an inconsistent way. In this case the changes have to be merged, or a clear priority (who will win) has to be established.
- The languages expressing the viewpoints may have been defined independent of each other, with *no dependency*. This probably means that it was discovered only *after the fact* that some parts of the model have to be synchronized. In this case the synchronization must be put into some kind of adapter language. It also means that the synchronization is not as clean as if it had been "designed into" the languages (see next item).
- In the mbeddr example, the information (the signature of the operation) was simply replicated, so the transformation was trivial. However, there may not be a 1:1 correspondence between the information in the two viewpoints. This makes the transformation more complex to write. In the worst case it may mean that the synchronization cannot be formally described and automated.

[53] The QVT-R transformation language has this capability, for example.

Sometimes the correspondence between models can only be expressed on an instance level (as in "this functional block corresponds to this software component")[54]. Consequently, developers have to manually express the correspondence (the trace links mentioned earlier). However, consistency checks (and possibly automatic synchronization) may still be possible, based on the manually expressed trace links.

[54] This often happens in the context of Progressive Refinement, as discussed in the previous subsection.

In my work with DSLs I have only encountered the simplest cases of synchronization, which is why we don't put much emphasis on this topic in the rest of the book. For more details see the papers by Diskin[55] and Stevens[56].

[55]
[56]

2.5 Completeness

Completeness[57] refers to the degree to which a language L is able to express programs that contain all information for them to be executed.

[57] This has nothing to do with Turing-completeness.

Let us introduce a function G ("code generator") that transforms a program p in L_D to a program q in L_{D-1}. For a complete language, p and q have the same semantics, i.e. $OB(p) == OB(G(p)) == OB(q)$ (see Section 2.3). For incomplete languages where $OB(G(p)) \subset OB(p)$ we have to write additional code in L_{D-1}, to obtain a program in D_{-1} that has the same semantics as intended by the original program in L_D. In cases where we use several viewpoints to represent various concerns of D, the set of fragments written for these concerns must be enough for complete D_{-1} generation.

Another way of stating this is that G produces a program in L_{D-1} that is not sufficient for a subsequent transformation (e.g., a compiler), only the manually written L_{D-1} code leads the sufficiency.

Embedded C: The Embedded C language is complete regarding D_{-1}, or even D_{-m} for higher levels of D, since higher levels are always built as extensions of its D_{-1}. Developers can always fall back to D_{-1} to express what is not expressible directly with L_D. Since the users of this system are developers, falling back to D_{-1} or even D_0 is not a problem. ◀

2.5.1 *Compensating for Incompleteness*

Integrating the L_{D-1} in case of an incomplete L_D language can be done in several ways:

- by calling "black box" code written in L_{D-1}. This requires concepts in L_D for calling D_{-1} foreign functions. No syntactic embedding of D_{-1} code is required, beyond the ability to call functions[58].

[58] In the simplest case, these functions don't even have arguments, so the syntax to call such a function is essentially just the function name.

- by directly embedding L_{D-1} code in the L_D program. This is useful if L_D is an extension of L_{D-1}, or if the tool provide adequate support for embedding the D_{-1} language into L_D programs. Note that L_{D-1} may not be analyzable, so mixing L_{D-1} into L_D code may compromise analyzability of the L_D code.

Just "pasting text into a text field", an approach used by several graphical modeling tools, is not productive, since no syntactic and semantic integration between the languages is provided. In most cases there is no tool support (syntax highlighting, code completion, error checking)

- by using composition mechanisms of L_{D-1} to "plug in" the manually written code into the generated code without actually modifying the generated files (also known as the Generation Gap pattern[59]). Example techniques for realizing this approach include generating a base class with abstract methods (requiring the user to implement them in a manually written subclass) or with empty callback methods which the user can use to customize in a subclass (for example, in user interfaces, you can return a position object for a widget, the default method returns **`null`**, default to the generic layout algorithm). You can delegate, implement interfaces, use **`#include`**, use reflection tricks, AOP or take a look at the well-known design patterns for inspiration. Some languages provide partial classes, where a class definition can be split over a generated file and a manually written file.

[59]

- or by inserting manually-written L_{D-1} code into the L_{D-1} code generated from the L_D program using protected regions. Protected regions are areas of the code, usually delimited by special comments, whose (manually written) contents are not overwritten during regeneration of the file

We strongly discourage the use of protected regions. You'll run into all kinds of problems: generated code is not a throw-away product anymore, you have to check it in, and you'll run into all kinds of funny situations with your version control system. Also, often you will accumulate a "sediment" of code that has been generated from model elements that are no longer in the model (if you don't use protected regions, you can delete the whole generated source directory from time to time, cleaning up the sediment). In the worst case, these may lead to compilation errors – even though the code is in fact not longer required.

For DSLs used by developers, incompleteness i usually not a problem because they are comfortable with writing the D_{-1} code in a programming language. Specifically, the DSL users are the same people as those who provide the remaining D_{-1} code, so coordination between the two roles is not a problem.

Component Architecture: This DSL is not complete. Only class skeleton and infrastructure integration code is generated from the models. The component implementation has to be implemented manually in Java using the Generation Gap pattern. The DSL is used by developers, so writing code in a subclass of a generated class is not a problem. ◀

For DSLs used by domain experts, the situation is different. Usually, they are not able to write D_{-1} code, so other people (developers) have to fill in the remaining concerns. Alternatively, developers can develop a predefined set of foreign functions that can be called from within the DSL. In effect, developers provide a standard library (cf. Section 2.1.2) which can be invoked as black boxes from DSL programs.

This requires elaborate collaboration schemes, because the domain experts have to communicate the remaining concerns via prose text or verbal communication.

WebDSL: The core of a web application is concerned with persistent data and their presentation. However, web applications need to perform additional duties outside that core, for which often useful libraries exist. WebDSL provides a *native interface* that allows a developer to call into a Java, library by declaring types and functions from the library in a WebDSL program. ◀

Note that a DSL that does not *cover* all of D can still be *complete*: not all of the programs imaginable in a domain may be expressed with a DSL, but those programs that can be expressed can be expressed completely, without any manually written code. Also, the code generated from a DSL program may require a framework written in L_{D-1} to run in. That framework represents aspects of D outside the scope of L_D.

Refrigerators: The cooling DSL only supports reactive, state based systems that make up the core of the cooling algorithm. The drivers used in the lower layers of the system, or the control algorithms controlling the actual compressors in the fridge,cannot be expressed with the DSL. However, these aspects are developed once and can be reused without adaptations, so using DSLs is not sensible. These parts are implemented manually in C. ◀

■ *Controlling D_{-1} Code* Allowing users to manually write D_{-1} code, and especially, if it is actually a GPL in D_0, comes with two additional challenges. Consider the following example: the generator generates an abstract class from some model element. The developer is expected to subclass the generated class and implement a couple of abstract methods. The manually written subclass needs to conform to a specific naming convention so some other generated code can instantiate the manually written subclass. The generator, however, just generates the base class and stops: how can you make sure developers actually do write that subclass, using the correct name[60]?

[60] Of course, if the constructor of the concrete subclass is called from another location of the generated code, and/or if the abstract methods are invoked, you'll get compiler errors. By their nature, they are on the abstraction level of the implementation code, however. It is not always obvious what the developer has to do in terms of the model or domain to get rid of these errors.

To address this issue, make sure there is there a way to make those conventions and idioms interactive. One way to do this is to generate checks/constraints *against the code base* and have them evaluated by the IDE, for example using Findbugs[61] or similar code checking tools. If one fails, an error message is reported to the developer. That error message can be worded by the developer of the DSL, helping the developer understand what exactly has to be done to solve the problem with the code.

[61] `http://findbugs.sourceforge.net/`

■ *Semantic Consistency* As part of the definition of a DSL you will implement constraints that validate the DSL program in order to ensure some property of the resulting system (see Section 2.3.1). For example, you might check dependencies between components in an architecture model to ensure components can be exchanged in the actual system. Of course such a validation is only useful if the manually written code does not introduce dependencies that are not present in the model. In that case the "green light" from the constraint check does not help much.

To ensure that promises made by the models are kept by the (manually written) code, use on of the following two approaches. First, generate code that does not allow violation of model promises. For example, don't expose a factory that allows components to look up and use any other component (creating dependencies), but rather use dependency injection to supply objects for the valid dependencies expressed in the model[62].

[62] A better approach is to build a *complete* DSL. The language used to express the behavior (which might otherwise plugged in manually in the generated code) is suitably limited and/or checked to enforce it does not lead to inconsistencies. This is a nice use case for language extension and embedding.

Component Architecture: The code generator to Java generates component implementation classes that use dependency injection to supply the targets for required ports. This way, the implementation class will have access to exactly those interfaces specified in the model. An alternative approach would be to simply hand to the implementation class some kind of factory or registry where a component implementation can look up instances of components that provide the interfaces specified by the required ports of the current component. However, this way it would be much harder to make sure that only those dependencies are accessed that are expressed in the model. Using dependency injection *enforces* this constraint in the implementation code. ◀

A second approach uses code checkers (like the Findbugs mentioned above) or architecture analysis tools to validate manually written code. You can easily generate the relevant checking rules for those tools from the models.

2.5.2 *Roundtrip Transformation*

Roundtrip transformation means that an L_D program can be recovered from a program in L_{D-1} (written from scratch, or changed manually after generation from a previous iteration of the L_D program). This is challenging, because it requires reconstituting the semantics of the L_D program from idioms or patterns used in the L_{D-1} code. This is the general reverse engineering problem and is not generally possible, although progress has been made over recent years (see for example[63]).

[63] ; ; and

Notice that the problem of "understanding" the semantics of a program written at a too-low abstraction level is the reason for DSLs in the first place: by providing linguistic abstractions for the relevant semantics, no "recovery" is necessary for meaningful analysis and transformation.

Note that for complete languages roundtripping is generally not useful, because the complete program can be written on L_D in the first place. Even if recovery of the semantics is possible it may not be practical: if the DSL provides significant abstraction over the L_{D-1} program, then the generated L_{D-1} program is so complicated, that manually changing the D_{-1} code in a consistent and correct way is tedious and error-prone.

Roundtripping has traditionally been used with respect to UML models and generated class skeletons. In that case, the abstractions between the model and the code are similar (classes), the tool basically just provides a different concrete syntax (diagrams). This similarity of abstractions in the code and the model made roundtripping possible to some extent. However, it also made the models relatively useless, because they did *not* provide a significant benefit in terms of abstraction over code details. We generally recommend to avoid (the attempt of building support for roundtripping.

Embedded C: This language does not support roundtripping, but since all DSLs are extensions of C, one can always add C code to the programs, alleviating the need for roundtripping in the first place. ◀

Refrigerators: Roundtripping is not required here, since the DSL is complete. The code generators are quite sophisticated, and nobody would want to manually change the generated C code. Since the DSL has proven to provide good coverage, the need to "tweak" the generated code has not come up. ◀

Component Architecture: Roundtripping is not supported. Changes to the interfaces, operation signatures or components have to be performed in the models. This has not been reported as a problem by the users, since both the implementation code and the DSL "look and feel" the same way – they are both Eclipse-based textual editors – and generation of the derived low level code happens automatically on saving a changed model. The workflow is seamless. ◀

Pension Plans: This is a typical application domain DSL where the users never see the generated Java code. Consequently, the language has to be complete and roundtripping is not useful and would not fit with the development process. ◀

2.6 *Language Modularity*

Reuse of modularized parts makes software development more efficient, since similar functionality does not have to be developed over and over again. A similar argument can be made for languages. Being able to reuse languages, or parts of languages, in new contexts makes designing DSLs more efficient.

Language modularization and reuse is often not driven by end user or domain requirements, but rather, by the experience of the language designers and implementers striving for consistency and avoidance of duplicate implementation work.

Language composition requires the composition of abstract syntax, concrete syntax, constraints/type systems and the execution semantics[64]. We discuss all of these aspect in this section. However, in the discussion of semantic integration, we consider only the case where the composed used the same (or closely related) behavioral paradigms[65], since, otherwise the composition can become very challenging. We mostly focus on imperative programs. We discuss behavioral paradigms in more detail in Section 3.

[64] It requires the composition of the IDE as well. However, with the language workbenches used in this book, this is *mostly* automatic.

[65] The behavioral paradigm is also known as the Model of Computation.

■ *Composition Techniques* We have identified the following four composition strategies: referencing, extension, reuse and embedding. We distinguish them regarding fragment structure and language dependencies, as illustrated in Fig. 2.25. Fig. 2.26 shows the relationships between fragments and languages in these cases[66].

[66] Note how in both cases the language definitions are modular: *invasive* modification of a language definition is not something we consider language modularity!

We consider these two criteria to be relevant for the following reasons. *Language dependencies* capture whether a language has to be designed with knowledge about a particular composition partner in mind in order to be composable with that partner. It is desirable in many scenarios that languages be composable *without* previous knowledge about all possible composition partners. *Fragment Structure* captures whether the two composed languages can be syntactically mixed, or whether separate viewpoints are used. Since modular concrete syntax can be a challenge, this is not always possible, though often desir-

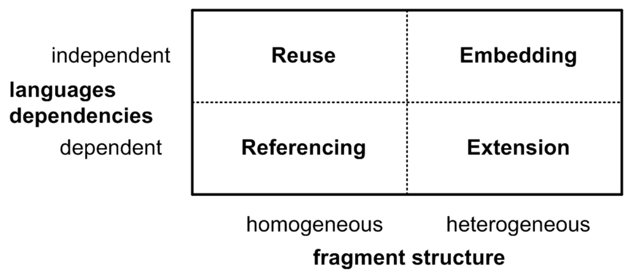


Figure 2.25: We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies.

able.

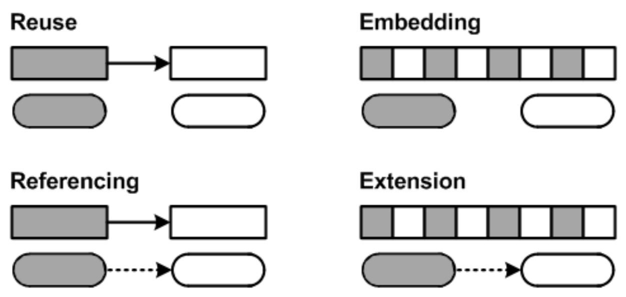


Figure 2.26: The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

■ *DSL Hell?* Reusing DSL also helps avoid the DSL Hell problem we discussed in the introduction. DSL hell refers to the danger that developers create new DSLs all the time, resulting in a large set of half-baked DSLs, each covering related domains, possibly with overlap, but still incompatible. Language modularization and reuse can help avoid this problem. Language extension allows users to add new language constructs to existing languages. They can reuse all the features of the existing language while still adding their own higher level abstractions. Language embedding lets language designers embed existing languages into new ones. This is particularly interesting in case of expression or query languages, that are relevant in many different contexts.

■ *More Detailed Examples* Part III of the book discusses the implementation of these modularization techniques with various tools (Section ??). As part of this discussion we present much more concrete and detailed examples of the various composition techniques. You may want to take a look at those examples while you read this section here.

2.6.1 Language Referencing

Language referencing (Fig. 2.27) enables *homogeneous* fragments with cross-references among them, using *dependent* languages.

A fragment f_2 depends on f_1. f_2 and f_1 are expressed with different languages l_2 and l_1. The referencing language l_2 depends on the referenced language l_1 because at least one concept in the l_2 references a concept from l_1. We call l_2 the *referencing* language, and l_1 the *referenced* language. While equations (1.2) and (1.3) (see Section 1.3) continue to hold, (1.1) does not. Instead

$$\forall r \in Refs_{l_2} \mid lo(r.from) = l_2 \ \wedge \ (lo(r.to) = l_1 \vee lo(r.to) = l_2) \qquad (2.1)$$

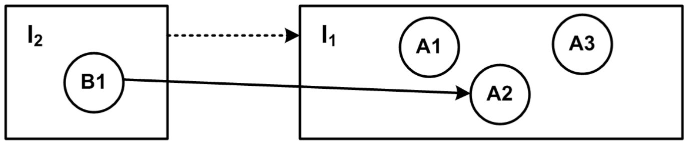


Figure 2.27: Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted arrows = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

■ *Viewpoints* As we have discussed before in Section 2.4, a domain D can be composed from different concerns. One way of dealing with this is to define separate concern-specific DSLs, each addressing one or more of the domain's concerns. A program then consists of a set of concern-specific fragments, that relate to each other in a well-defined way using language referencing. This approach has the advantage that different stakeholders can modify "their" concern independent of others. It also allows reuse of the independent fragments and languages with different referencing languages. The obvious drawback is that for tightly integrated concerns the separation into separate fragments can be a usability problem.

Referencing implies knowledge about the relationships of the languages as they are designed. Viewpoints are the classical case for this. The dependent languages *cannot* be reused, because of the dependency on the other language.

Refrigerators: As an example, consider the domain of refrigerator configuration. The domain consists of three concerns. The first concern H describes the hardware structure of refrigerators appliances including compartments, compressors, fans, valves and thermometers. The second concern A describes the cooling algorithm using a state-based, asynchronous language. Cooling programs refer to hardware building blocks and access the their properties in expressions and commands. The third concern is testing T. A cooling test can test and simulate cooling programs. The dependencies are as follows: $A \rightarrow H$ and $T \rightarrow A$. Each of these concerns are implemented as a separate language with references between them. H and A are separated because H is defined by product management, whereas A is defined by thermodynamicists. Also, several algorithms for the same hardware must be supported, which makes separate fragments for H and A useful. T is separate from A because tests are not strictly part of the product definition and may be enhanced after a product has been released. These languages have been built as part of a single project, so the dependencies between them are not a problem. ◀

■ *Progressive Refinement* Progressive refinement, also introduced earlier (Section 2.4.6), also makes use of language referencing.

2.6.2 *Language Extension*

Language extension Fig. 2.28 enables *heterogeneous* fragments with *dependent* languages. A language l_2 extending l_1 adds additional language concepts to those of l_1. We call l_2 the *extending* language (or language extension), and l_1 the *base* language. To allow the new concepts to be used in the context provided by l_1, some of them extend concepts in l_1. So, while l_1 remains independent, l_2 becomes dependent on l_1 since

$$\exists i \in Inh(l_2) \mid i.sub = l_2 \ \wedge \ i.super = l_1 \tag{2.2}$$

Consequently, a fragment f contains language concepts from both l_1 and l_2:

$$\forall e \in E_f \mid lo(e) = l_1 \vee lo(e) = l_2 \tag{2.3}$$

In other words, $C_f \subset (C_{l_1} \cup C_{l_2})$, so f is *heterogeneous*. For heterogeneous fragments (1.3) does not hold anymore, since

$$\begin{aligned}\forall c \in Cdn_f \mid (lo(co(c.parent)) = l_1 \vee lo(co(c.parent)) = l_2) \wedge \\ (lo(co(c.child)) = l_1 \vee lo(co(c.child)) = l_2)\end{aligned} \tag{2.4}$$

Note that copying a language definition and changing it does not constitute a case of language extension, because the extension is not modular, it is invasive. Also, a native interfaces that support calling one language from another one (like calling C from Perl or Java) is not language extension; rather it is a form of language referencing. The fragments remain homogeneous.

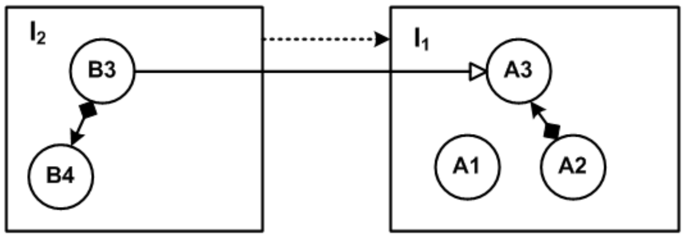


Figure 2.28: Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, just like $A3$. This plugs l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

Language extension is especially interesting if D_0 languages are extended, making a DSL an extension of a general purpose language.

Language extension fits well with the hierarchical domains introduced in Section ??: a language L_B for a domain D may extend a language L_A for D_{-1}. L_B contains concepts specific to D, making analysis and transformation of those concepts possible without pattern matching and semantics recovery. As explained in the introduction, the new concepts are often reified from the idioms and patterns used when using an L_A for D. Language semantics are typically defined by mapping the new abstractions to just these idioms (see Section 2.3) *inline*. This process, also known as *assimilation*, transforms a heterogeneous fragment (expressed in L_D and L_{D+1}) into a homogeneous fragment expressed only with L_D.

Extension is especially useful for bottom-up domains. The common patterns and idioms identified for a domain can be reified directly into linguistic abstractions, and used directly in the language from which they have been embedded. Incomplete languages are not a problem, since users can easily fall back to D_{-1} to implement the rest. Since DSL users see the D_{-1} code all the time anyway, they will be comfortable falling back to D_{-1} in exceptional cases. This makes extensions suitable only for DSLs used by developers. Domain expert DSLs are typically not implemented as extensions.

Embedded C: As an example consider embedded programming. The C programming language is typically used as the GPL for D_0 in this case. Extensions for embedded programming include state machines, tasks or data types with physical units. Language extensions for the subdomain of real-time systems may include ways of specifying deterministic scheduling and worst-case execution time. For the avionics subdomain support for remote communication using some of the bus systems used in avionics could be added. ◀

Extension comes in two flavors. One really feels like extension, and the other one feels more like embedding.

Extension-Flavor In the first case we provide (a little, local) additional syntax to an otherwise unchanged language. For example, C may be extended with new data types and literals for complex numbers as in `complex c = (3+2i);`. The programs still essentially look like C programs (or few) particular places, something is different.

Embedding Flavor The other case is where we create a completely new language, but reuse some of the syntax provided by the base language. For example, we could create a state machine language that reuses C's expression and types in guard conditions. This use case *feels* like embedding (we embed syntax from the base language in our new language), but in the classification according to syntactic integration and dependencies, it is still extension. Embedding would prevent dependencies between the state machine language and C.

The embedding flavour *is* often suitable for use with DSLs that are used by non-programmers, since the "embedded" subset of the language is often small and simple to understand. Once again, expression languages are the prime example for this.

Language extension is also a very useful way to address the problem that DSLs often start simple, but then become more complicated over time, because new corners or intricacies in the domain are discovered as users gain more experience in the domain. These corner cases and intricacies can be factored into a separate language module that extends the core DSL. The use of these extensions can then initially be restricted to a few users in order to find out if they are really needed. Different experiments can even be performed at the same time, with different groups of users using different extensions. Even once these extensions have proven useful, "advanced" language feature are restrictable this way to a small group of "advanced" users who handle the hard cases by using the extension.

Incremental extension can help avoid the feared customization cliff. The customization cliff is a term introduced by Steve Cook[67]: *once you step outside of what is covered by your DSL, you plunge down a cliff onto the rocks of the low-level platform.* If DSLs are built as incremental extensions of the next lower language, then stepping outside any DSL on level D will only plunge you down to the language for D_{-1}. And presumably you can always create an additional extension that extends your DSL to cover an additional, initially unexpected aspect.

[67] `http://bit.ly/yzZb1u`

Defining a D languages as an extension of a D_{-1} language can also have drawbacks. The language is tightly bound to the D_{-1} language it is extended from. While it is possible for a standalone DSL in D to generate implementations for different D_{-1} languages, this is not easily possible for DSLs that are extensions of a D_{-1} language. Also, interaction with the D_{-1} language may make meaningful semantic analysis of complete programs (using L_D and L_{D-1} concepts) hard. This problem can be limited if isolated L_D sections are used in which interaction with L_{D-1} concepts is limited and well-defined. These isolated sections remain analyzable.

Restriction is often useful for the embedding-flavor of extension. For example, when embedding C expressions into the state machine language, we may want to restrict users from using the pointer-related expressions.

■ *Restriction* Sometimes language extension is also used to *restrict* the set of language constructs available in the subdomain. For example, the real-time extensions for C may restrict the use of dynamic memory allocation, the extension for safety-critical systems may prevent the use of **`void`** pointers and certain casts. Although the extending language is in some sense smaller than the extended one, we still consider this a case of language extension, for two reasons. First, the restrictions are often implemented by *adding additional* constraints that report errors if the restricted language constructs are used. Second, a marker concept may be added to the base language. The restriction rules are then enforced for children of these marker concepts (e.g., in a module marked as "safe", one cannot use void pointers and the prohibited casts).

Embedded C: Modules can be marked as *MISRA-compliant*, which prevents the use of those C constructs that are not allowed in MISRA-C[68]. Prohibited concepts are reported as errors directly in the program. ◀

[68] `http://bit.ly/HpW4xV`

2.6.3 *Language Reuse*

Language reuse (Fig. 2.29) enables *homogenous* fragments with *independent* languages. Given are two independent languages l_2 and l_1 and two fragment f_2 and f_1. f_2 depends on f_1, so that

$$\exists r \in Refs_{f_2} \mid fo(r.from) = f_2 \wedge \\ (fo(r.to) = f_1 \vee fo(r.to) = f_2) \tag{2.5}$$

One could argue that in this case reuse is just a clever combination of referencing and extension. While this is true from an implementation perspective, it is worth describing as a separate approach, because it enables the combination of two *independent languages* by adding an adapter *after the fact*, so no pre-planning during the design of l_1 and l_2 is necessary.

Since l_2 is independent, it cannot directly reference concepts in l_1. This makes l_2 reusable with different languages (in contrast to language referencing, where concepts in l_2 reference concepts in l_1). We call l_2 the *context* language and l_1 the *reused* language.

One way of realizing dependent fragments while retaining independent languages is using an adapter language l_A where l_A *extends* l_2 and

$$\exists r \in Refs_{l_A} \mid lo(r.from) = l_A \ \wedge \ lo(r.to) = l_1 \tag{2.6}$$

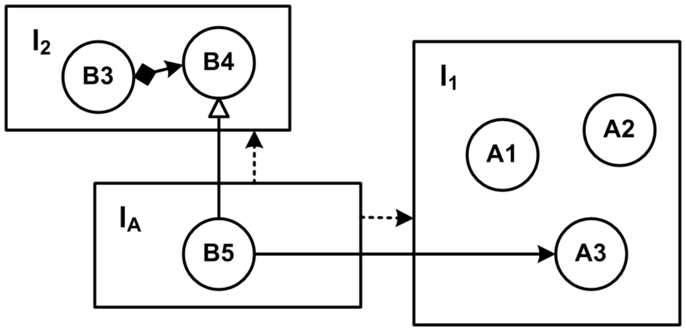


Figure 2.29: Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

While language referencing supports reuse of the referenced language, language reuse supports the reuse of the *referencing* language as well. This makes sense for concern DSLs that have the potential to be reused in many domains, with minor adjustments. Examples include role-based access control, relational database mappings and UI specification.

Example: Consider a language for describing user interfaces. It provides language concepts for various widgets, layout definition and disable/enable strategies. It also supports data binding, where data structures are associated with widgets, to enable two-way synchronization between the UI and the data. Using language reuse, the same UI language can be used with *different* data description languages. Referencing would not achieve this goal because the UI language would have a direct dependency on a particular data description language. Changing the dependency direction to $data \rightarrow ui$ doesn't solve the problem either, because this would go against the generally accepted idiom that UI has dependencies to the data, but not vice versa (cf. the MVC pattern). ◀

Generally, the referencing language is built with the knowledge that it will be reused with other languages, so hooks may be provided for adapter languages to plug in.

Example: The UI language thus may define an abstract concept `DataMapping` which is then extended by various adapter languages. ◀

2.6.4 *Language Embedding*

Language embedding (Fig. 2.30) enables *heterogeneous* fragments with *independent* languages. It is similar to reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now embed instances of concepts from l_2 in a fragment f expressed with l_1, so

$$\begin{aligned}\forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge \\ (lo(co(c.child)) = l_1 \vee lo(co(c.child)) = l_2))\end{aligned} \quad (2.7)$$

Unlike language extension, where l_2 depends on l_1 because concepts in l_2 extends concepts in l_1, there is no such dependency in this case. Both languages are independent. We call l_2 the *embedded* language and l_1 the *host* language. Again, an adapter language l_A that extends l_1 can be used to achieve this, where

$$\exists c \in Cdn_{l_A} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \quad (2.8)$$

Embedding supports syntactic composition of independently developed languages. As an example, consider a state machine language that can be combined with any number of programming languages such as Java or C. If the state machine language is used together with Java, then the guard conditions used in the transitions should be Java expressions. If it is used with C, then the expressions should be C expressions. The two expression languages, or in fact, any other one,

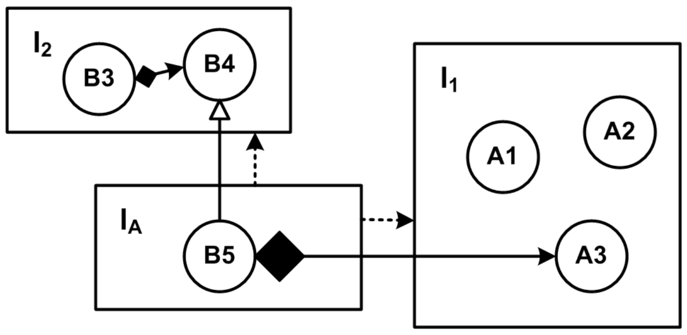


Figure 2.30: Embedding: l_1 and l_2 are independent languages. However, we still want to use them in the same fragment. To enable this, an adapter language l_A is added. It depends on both l_1 and l_2, and uses inheritance and composition to adapt l_1 to l_2 (this is the almost the same structure as in the case of reuse; the difference is that $B5$ now *contains* $A3$, instead of just referencing it.)

must be embeddable in the guard conditions. So the state machine language cannot depend on any particular expression language, and the expression languages of C or Java obviously cannot be designed with knowledge about the state machine language. Both have to remain independent, and have to be embedded using an adapter language.

Note that if the state machine language is specifically built to "embed" C expressions, then this is a case of Language Extension, since the state machine language depends on the C expression language.

Another example is embedding a database query language such as Linq or SQL in *different* programming languages (Java, C#, C). Again, the query language may not have a dependency on any programming language (otherwise it would not be embeddable in all of them). The problem could be solved by extension (with embedding flavor), but then the programming language would have to be invasively changed – it now has to get a dependency on the query language. Using embedding, this dependency can be avoided.

When embedding a language, the embedded language must often be extended as well. In the state machine example, new kinds of expressions must be added to support referencing event parameters defined in the host language. In case of the query language, method arguments and local variables should probably me usable as part of the queries (`... WHERE somecolumn = someMethodArg`). These additional expressions will typically reside in the adapter language as well.

Just as in the embedding-flavored extension case (cf. Section 2.6.2), sometimes the embedded language must also be restricted. If you embed the C expression language in state machine guard conditions, you may want to restrict the user from using pointer types or all the expressions related to pointers in C.

WebDSL: In order to support queries over persistent data, WebDSL embeds the Hibernate Query Language (HQL) such that HQL queries can be used as expressions. Queries can refer to entity declarations in the program and to variables in the scope of the query. ◀

Pension Plans: The pension workbench DSL embeds a spreadsheet language for expressing unit tests for pension plan calculation rules. The spreadsheet language comes with its own simple expression language to be used inside the cells. A new expression as been added to reference pension rule input parameters so they can be used inside the cells. ◀

■ *Cross-Cutting Embedding, Meta Data* A special case of embedding is handling meta data. We define meta data as program elements that are not essential to the semantics of the program, and are typically not handled by the primary model processor. Nonetheless these data must relate to program elements, and, at least from a user's perspective, they often need to be embedded in programs. Since most of them are rather generic, embedding is the right composition mechanism: no dependency to any specific language should be necessary, and the meta data should be embeddable in any language. Example meta data includes:

Documentation should be attachable to any program element, and in the documentation text, other program elements should be referencable.

Traces capture typed relationships between program elements or between program elements and requirements or other documentation ("this program element *implements* that requirement")

Presence Conditions in product line engineering describe whether a program element should be available in the program for a given product configuration ("this procedure is only in the program in the *international* variant of the product").

In projectional editors, this meta data can be stored in the program tree, and shown only optionally, if some global configuration switch is `true`. In textual editors, meta data is often stored in separate files, using pointers to refer to the respective model elements. The data may be shown in hovers or views adjacent to the editor itself.

Embedded C: The system supports various kinds of meta data, including traces to requirements and documentation. They are implemented with MPS' *attribute* mechanism which is discussed in part on MPS in Section **??**. As a consequence of how MPS attributes work, these meta data can be applied to program elements defined in any arbitrary language. ◀

2.6.5 Implementation Challenges and Solutions

The previous subsections discussed four strategies for language composition. In this section we describe some of the challenges regarding syntax, type systems and transformations for these four strategies.

■ *Syntax* Referencing and Reuse keeps fragments homogeneous. Mixing of concrete syntax is not required. A reference between fragments is usually simply an identifier and does not have its own internal structure for which a grammar would be required[69]. The name resolution phase can then create the actual cross-reference between abstract syntax objects.

[69] Sometimes the reference use qualified names, in which case the strings use dots and colons. However, this is still a trivial token structure, so it is acceptable to define the structure separately in both languages.

Refrigerators: The algorithm language contains cross-references into the hardware language. Those references are simple, dotted names such as `compartment1.valve`. ◄

Example: In the UI example, the adapter language simply introduces dotted names to refer to fields of data structures. ◄

Extension and Embedding requires modular concrete syntax definitions because additional language elements must be "mixed" with programs written with the base/host language. As we discuss in Part III (mostly in Section **??**), combining independently developed languages after the fact can be a problem: depending on the parser technology, the combined grammar may not be parsable with the parser technology at hand. There are parser technologies that do not exhibit this problem, and projectional editors avoid it by definition. However, several widely used language workbenches have problems in this respect. For more details.

Embedded C: State machines are hosted in regular C programs. This works because the C language's `Module` construct contains a collection of `IModuleContents`, and the `StateMachine` concept implements the `IModuleContent` concept interface. This state machine language is designed specifically for being embedded into C, so it can access and extend `IModuleContent` (Fig. 2.31). If the state machine language were embeddable in any host language in addition to C, this dependency on `ModuleContent` (from the C base language) would not be allowed. An adapter language would have to be created which adapts a `StateMachine` to `IModuleContent`. ◄

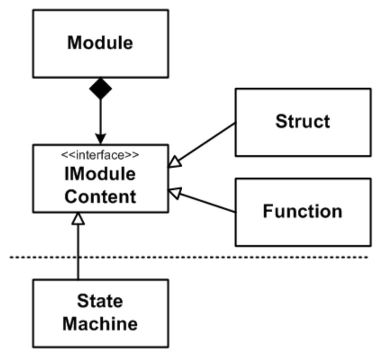


Figure 2.31: The core language (above the dotted line) defines an interface `IModuleContent`. Anything that should be hosted inside a `Module` has to implement this interface, typically from another language. `StateMachines` are an example.

■ *Type Systems* For Referencing, the type system rules and constraints of the referencing language typically have to take into account the referenced language. Since the referenced language is known when developing the referencing language, the type system can be implemented with the referenced language in mind as well.

Refrigerators: In the refrigerator example, the algorithm language defines typing rules for hardware elements (from the hardware language), because these types are used to determine which properties can be accessed on the hardware elements (e.g., a compressor has a property `active` that controls if it is turned on or off). ◀

In case of Extension, the type systems of the base language must be designed in a way that allows adding new typing rules in language extensions. For example, if the base language defines typing rules for binary operators, and the extension language defines new types, then those typing rules may have to be overridden to allow the use of existing operators with the new types.

Embedded C: A language extension provides types with physical units (as in `100 kg`). Additional typing rules are needed to override the typing rules for C's basic operators (+, -, *, /, etc.). MPS supports declarative type system specification, so you can just *add* additional typing rules for the case where one or both of the arguments have a type with a physical unit. ◀

For Reuse and Embedding, the typing rules that affect the interplay between the two languages reside in the adapter language. The type systems of both languages must be extensible in the way described in the previous paragraph on Extension.

Example: In the UI example the adapter language will have to adapt the data types of the fields in the data description to the types the UI widgets expect. For example, a combo box widget can only be bound to fields that have some kind of text or enum data type. Since the specific types are specific to the data description language (which is unknown at the time of creation of the UI language), a mapping must be provided in the adapter language. ◀

■ *Transformation* In this section we use the terms *transformation* and *generation* interchangably. In general, the term transformation is used if one tree of program elements is mapped to another tree, while generation describes the case of creating text from program trees. However, for the discussions in this section, this distinction is generally not relevant.

Three cases have to be considered for Referencing. The first one (Fig. 2.32) propagates the referencing structure to the target fragments. We call these two transformations *single-sourced*, since each of them only uses a single, homogeneous fragment as input and creates a single, homogeneous fragment as output, typically with references between them. Since the referencing language is created with the knowledge about the referenced language, the generator for the referencing language can be written with knowledge about the names of the elements that have to be referenced in the fragment generated from the referenced fragment. If a generator for the referenced language already exists, it can be reused unchanged. The two generators basically

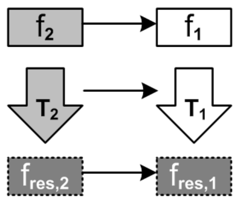


Figure 2.32: Referencing: Two separate, dependent, single-source transformations

share knowledge about the names of generated elements.

Component Architecture: In the types viewpoint, interfaces and components are defined. The types viewpoint is independent, and it is sufficient for the generation of the code necessary for implementing component behavior: Java base classes are generated that act as the component implementations (expected to be extended by manually written subclasses). A second, dependent viewpoints describes component instances and their connections;it depends on the types viewpoint. A third one describes the deployment of the instances to execution nodes (servers, essentially). The generator for the deployment viewpoint generates code that actually instantiates the classes that implement components, so it has to know the names of those generated (and hand-written) classes. ◀

The second case (Fig. 2.33) is a multi-sourced transformation that creates one single homogeneous fragment. This typically occurs if the referencing fragment is used to guide the transformation of the referenced fragment, for example by specifying transformation strategies (annotation models). In this case, a new transformation has to be written that takes the referencing fragment into account. The possibly existing generator for the referenced language cannot be reused as is.

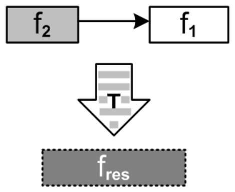


Figure 2.33: A single multi-sourced transformation.

Refrigerators: The refrigerator example uses this case. The code generator that generates the C code that implements the cooling algorithm takes into account the information from the hardware description model. A single fragment is generated from the two input models. The generated code is C-only, so the fragment remains homogeneous. ◀

The third case, an alternative to rewriting the generator, is the use of a preprocessing transformation (Fig. 2.34), that changes the referenced fragment in a way consistent with what the referencing fragment prescribes. The existing transformations for the referenced fragment can then be reused.

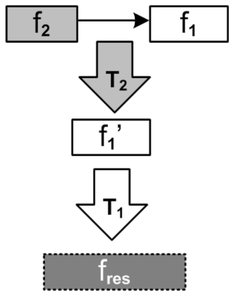


Figure 2.34: A preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment

As we have discussed above, language extensions are usually created by defining linguistic abstractions for common idioms of a domain D. A generator for the new language concepts can simply recreate those idioms when mapping L_D to L_{D-1}, a process also called assimilation. In other words, transformations for language extensions map a heterogeneous fragment (containing L_{D-1} and L_D code) to a homogeneous fragment that contains only L_{D-1} code (Fig. 2.35). In some cases additional files may be generated, often configuration files. In any case, the subsequent transformations for L_{D-1}, if any, can be reused unchanged.

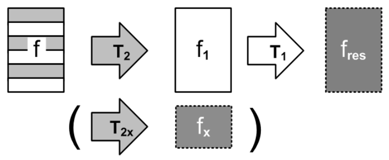


Figure 2.35: Extension: transformation usually happens by assimilation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often configuration files.

Embedded C: State machines are generated down to a function that contains a `switch` statement, as well as `enum`s for states and events. Then the existing C-to-text transformations are reused unchanged. In addition, the state machines are also transformed into a dot file that is used to render the state machine graphically via graphviz. ◀

Sometimes a language extension requires rewriting transformations defined by the base language. In this case, the transformation engine must support *overriding* transformations by transformations defined in another language.

Embedded C: In the data-types-with-physical-units example, the language also provides range checking and overflow detection. So if two such quantities are added, the addition is transformed into a call to a special `add` function instead of using the regular plus operator. This function performs overflow checking and addition. MPS supports transformation priorities that can be used to override the existing transformation with a new one. ◀

Language Extension introduces the risk of semantic interactions. The transformations associated with several independently developed extensions of the same base language may interact with each other. To avoid the problem, transformations should be built in a way so that they do not "consume scarce resources" such as inheritance links.[70]

[70] It would be nice if DSL tools would detect such conflicts statically, or at least supported a way of marking two languages or extensions as *incompatible*. However, none of the tools I know support such features.

Example: Consider the (somewhat constructed) example of two extensions to Java that each define a new statement. When assimilated to pure Java, both new statements require the surrounding Java class to extend a specific, but different base class. This won't work because a Java class can only extend one base class. ◀

Interactions may also be more subtle and affect memory usage or execution performance. Note that this problem is not specific to languages, it can occur whenever several independent extensions of a something can be used together, ad hoc. A more thorough discussion of the problem of semantic interactions is beyond the scope of this book.

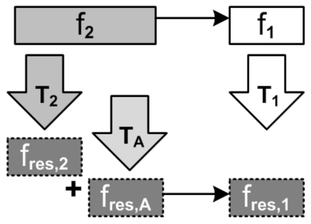


Figure 2.36: Reuse: Reuse of existing transformations for both fragments plus generation of adapter code

In the Reuse scenario, it is likely that both the reused and the context language already come with their own generators. If these generators transform to different, incompatible target languages, no reuse is possible. If they transform to a common target languages (such as Java or C) then the potential for reusing previously existing transformations exists.

There are three cases to consider. The first one, illustrated in Fig. 2.36, describes the case where there is an existing transformation for the reused fragment and an existing transformation for the context frag-

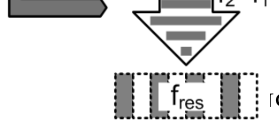


Figure 2.37: Reuse: composing transformations

ment – the latter being written with the knowledge that later extension will be necessary. In this case, the generator for the adapter language may "fill in the holes" left by the reusable generator for the context language. For example, the generator of the context language may generate a class with abstract methods; the adapter may generate a subclass and implement these abstract methods.

In the second case, Fig. 2.37, the existing generator for the reused fragment has to be enhanced with transformation code specific to the context language. A mechanism for composing transformations is needed.

The third case, Fig. 2.38, leaves composition to the target languages. We generate three different independent, homogeneous fragments, and a some kind of weaver composes them into one final, heterogeneous artifact. Often, the weaving specification is the intermediate result generated from the adapter language. An example implementation could use AspectJ.

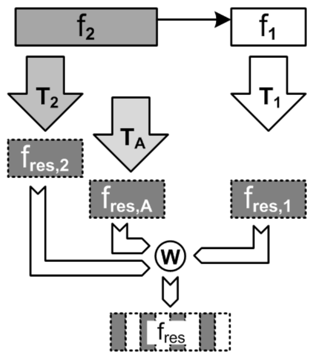


Figure 2.38: Reuse: generating separate artifacts plus a weaving specification

An embeddable language may not come with its own generator, since, at the time of implementing the embeddable language, one cannot know what to generate – its purpose is to be embedded! In that case, when embedding the language, a suitable generator has to be developed. It will typically either generate host language code (similar to generators in the case of language extension) or directly generate to the same target language that is generated to by the host language.

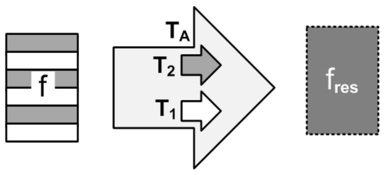


Figure 2.39: In transforming embedded languages, a new transformation has to be written if the embedded language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the emebedded languages.

If the embeddable language comes with a generator that transforms to the same target language as the embedding language, then the generator for the adapter language can coordinate the two, and make sure a single, consistent fragment is generated. Fig. 2.39 illustrates this case.

Just as language extension, language embedding may also lead to semantic interactions if multiple languages are embedded into the same host language.

2.7 *Concrete Syntax*

A good choice of concrete syntax is important for DSLs to be accepted by the intended user community. Especially (but not exclusively) in business domains, a DSL will only be successful if and when it uses notations that directly fit the domain – there might even be existing, established notations that should be reused. A good notation makes expression of common concerns simple and concise and provides sensible defaults. It is acceptable for less common concerns to require a bit more verbosity in the notation.

2.8 Design Concerns for Concrete Syntax

In particular the following concerns may be addressed when designing a concrete syntax[71]:

[71] These concerns do not just depend on the concrete syntax, but also on the abstract syntax and the expressiveness of the language itself (which is discussed in Section 2.1). However, the concrete syntax has a major influence, which is why we discuss it here.)

Writability A writable syntax is one that can be written efficiently. This usually means that the syntax is concise, because users have to type less. However, a related aspect is also tool support: the degree to which the IDE can provide better editing support[72] (code completion and quick fixes in particular) makes a difference for readability.

[72] See the example of the `select` statement in

Readability A readable syntax means that it can be read effectively. A more concise syntax is not necessarily more readable, because context may be missing[73], in particular for people other than those who have written the code.

[73] A good example of this dilemma are the APL or M languages: the syntax is so concise that it is really hard to read.

Learnability A learnable syntax is useful to newbies, in particular because it can be "explored", often exploiting IDE support[74]. For example, the more the language uses concepts that have a direct meaning in the domain, the easier it is for domain users to lean the language.

[74] "Just press Ctrl-Space and the tool will tell you what you can type next"

Effectiveness Effectivenss relates to the degree that a language enables routined users to effectively express typical domain problems *after* they have learned the language.

■ *Tradeoffs* It is obvious that some of these concerns are in conflict. A very writable language may not be very readable. If a group of stakeholders `R` uses artifacts developed by another group `W` (e.g. by referencing some of the program elements) it is important a readable language is used. A learnable language may feel "annoyingly verbose and cumbersome" to routined users after a while[75]. However, creating a effective syntax and trying to convince the users to adopt the language although it is hard(er) to learn may be a challenge.

[75] Note that if a specific DSL is only used irregularly, then users probably never become routined, and have to re-learn the language each time they use it.

For DSLs whose programs have a short lifetime (as in scripting languages) readability is often not very important, because the programs are thrown away once they have performed there particular task.

■ *Multiple Notations* One way to solve these dilemmas is to provide different concrete syntaxes for the same abstract syntax, and let users choose (beginners chose a more learnable one and switch to a more effective one over time). However, depending on the tooling used, this can be a lot of work.

■ *Multiple Notations* For projectional editors it is relatively easy to define several notations for the same language concept. By changing

We have the equivalent of multiple notations for the same language in the real world. English can be spoken, written, transported via morse code or even expressed via sign language. Each of these is optimized for certain contexts and audiences.

the projection rules, existing programs can be shown in a different way. In addition, different notations (possibly showing different concerns of the overall program) can be used for different stakeholders.

Embedded C: For state machines, the primary syntax is textual. However, a tabular notation is supported as well. The projection can be changed as the program is edited, rendering the same state machine textually or as a table. As mentioned above, a graphical notation will be added in the future, as MPS' support for graphical notations improves. ◄

Refrigerators: The refrigerator DSL uses graphical visualizations to render diagrams of the hardware structure, as well as a graphical state charts representing the underlying state machine. ◄

Another option to resolve the leanability vs. effectiveness dilemma is to create an effective syntax and help new users by good documentation, training and/or IDE support (templates, wizards).

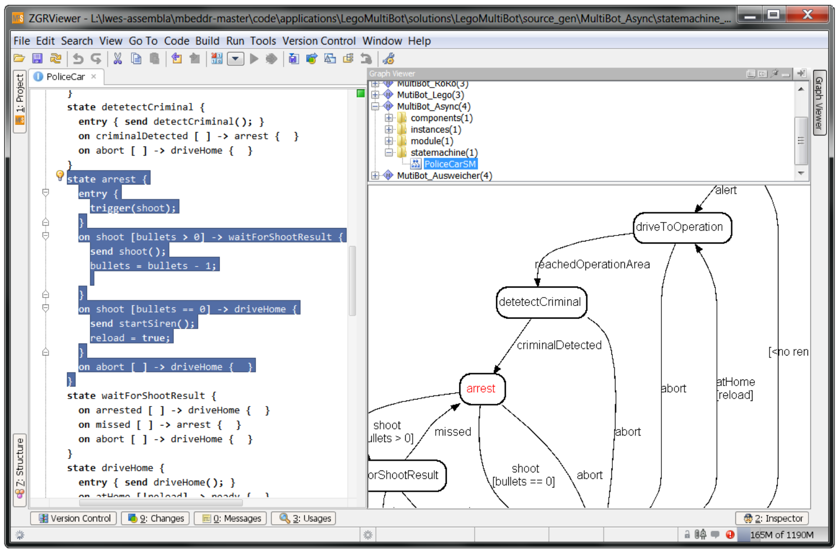
■ *Reports and Visualization* A visualization is a graphical representation of a model that cannot be edited. It is created from the core model using some kind of transformation, and highlights a particular aspect of the source program. It is often automatically laid out (possibly with layouting hints from the user). The resulting diagram may be static (i.e. an image file is generated) or interactive (where users can show, hide and focus on different parts of the diagram). It may provide drill-down back to the core program (double-clicking on the figure in the image opens the code editor at the respective location.)[76].

[76] Graphviz is one of the most well-known tools for this kind of visualization. Another one is Jan Koehnlein's Generic Graph View at `http://bit.ly/HpW4xV`

A report has the same goals (highlighting a particular aspect of the source program, while not being editable) but uses a textual notation.

Visualizations and reports are a good way of resolving a potential conflict where the primary DSL users want to use a writability notation and other stakeholders still want a more readable representation. Since reports and visualizations are not the primary notation, it is possible to create several different visualizations or reports for the sourve program, highlighting different aspects of the core program.

Embedded C: In the mbeddr components extension, we support several notations. The first one shows interfaces and the components that provide and require these interfaces. The second one shows component instances and the connections between their provided and required ports. Finally, there is a third visualization that applies to all mbeddr models, not just those that use components: it shows the modules, their imports (i.e. module dependencies) as well as the public contents of these modules (functions, structs, components, test cases). ◄



```
    }
    state detetectCriminal {
      entry { send detectCriminal(); }
      on criminalDetected [ ] -> arrest {   }
      on abort [ ] -> driveHome {   }
    }
    state arrest {
      entry {
        trigger(shoot);
      }
      on shoot [bullets > 0] -> waitForShootResult {
        send shoot();
        bullets = bullets - 1;
        
      }
      on shoot [bullets == 0] -> driveHome {
        send startSiren();
        reload = true;
      }
      on abort [ ] -> driveHome {   }
    }
    state waitForShootResult {
      on arrested [ ] -> driveHome {   }
      on missed [ ] -> arrest {   }
      on abort [ ] -> driveHome {   }
    }
    state driveHome {
      entry { send driveHome(); }
```

Figure 2.40: mbeddr C also supports graphical visualizations of state machines. For every state machine, a dot representation is automatically generated. The image is then rendered by graphviz directly in the IDE. Double-clicking on a state selects the respective program element in the editor.

2.9 *Classes of Concrete Syntax*

There are a couple of major classes for DSL concrete syntax[77]: *Textual* DSLs use linear textual notations, typically based on ASCII or Unicode characters. They basically look and feel like traditional programming languages. *Graphical* DSLs use graphical shapes. An important subgroup is represented by those that use box-and-line diagrams that look and feel like UML class diagrams or state machines. However, there are more options for graphical notations, such as those illustrated by UML timing diagrams or sequence diagrams. *Symbolic* DSLs are textual DSLs with an extended set of symbols, such as fraction bars, mathematical symbols or subscript and superscript. *Tables and Matrices* are a powerful way to represent certain kinds of data and can play an important part for DSLs.

[77] Sometimes form-based GUIs or trees views are considered DSLs. I disagree, because this would make any GUI application a DSL.

The perfect DSL tool should support freely combining and integrating the various classes of concrete syntax, and be able to show (aspects of) the same model in different notations. As a consequence of tool limitations, this is not always possible, however. The requirements for concrete syntax are a major driver in tool selection.

■ *When to Use Which Form* We do not want to make this section a complete discussion between graphical and textual DSLs – a discussion, that is often heavily biased by previous experience, prejudice and tool capabilities. Here are some rules of thumb. Purely textual DSLs integrate very well with existing development infrastructures, making their adoption relatively easy. They are very well suited for detailed descriptions, anything that is algorithmic or generally resembles (tra-

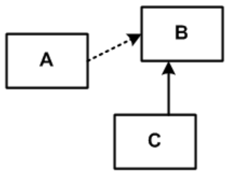


Figure 2.41: Graphical Notation for Relationships

ditional) program source code. A good textual syntax can be very effective (in terms of the design concerns discussed above). Symbolic notations can be considered "better textual", and lend themselves to domains that make heavy use of symbols and special notations; many scientific and mathematical domains come to mind. Tables are very useful for collections of similarly structured data items, or for expressing how two independent dimensions of data relate. Tables emphasize readability over writability. Finally, graphical notations are very good for describing relationships (Fig. 2.41), flow (Fig. 2.42) or timing and causal relationships (Fig. 2.43). They are often considered easier to learn, but may be perceived as less effective by experienced users.

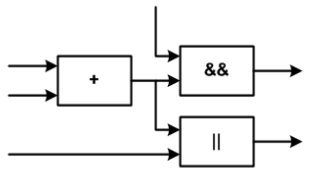


Figure 2.42: Graphical Notation for Flow

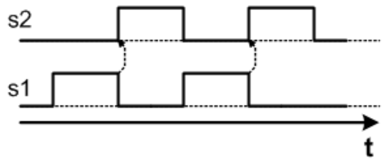
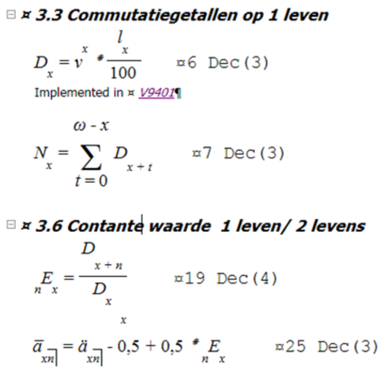


Figure 2.43: Graphical Notation for Causality and Timing


⊟ ¤ *3.3 Commutatiegetallen op 1 leven*

$D_x = v^x * \frac{l_x}{100}$ ¤6 Dec(3)

Implemented in ¤ V9401¶

$N_x = \sum_{t=0}^{\omega - x} D_{x+t}$ ¤7 Dec(3)

⊟ ¤ *3.6 Contante waarde 1 leven/ 2 levens*

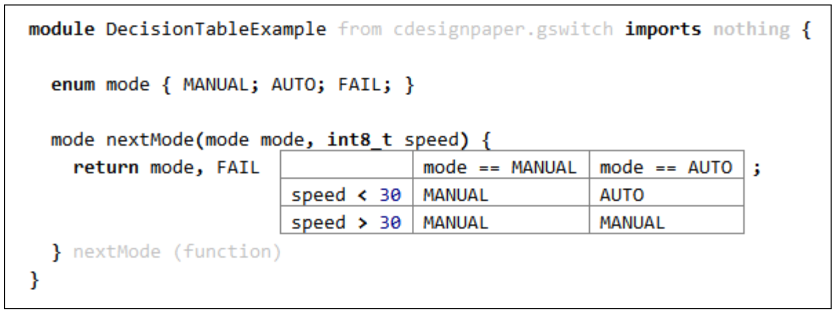
$_nE_x = \frac{D_{x+n}}{D_x}$ ¤19 Dec(4)

$\bar{a}_{x\overline{n}|} = \ddot{a}_{x\overline{n}|} - 0{,}5 + 0{,}5 * {}_nE_x$ ¤25 Dec(3)

Figure 2.44: Mathematical notations used to express insurance math in the pension workbench.

Pension Plans: The pension DSL uses mathematical symbols and notations to express insurance mathematics (Fig. 2.44). A table notation is embedded to express unit tests for the pension plan calculation rules. A graphical projection shows dependencies and specialization relationships between plans. ◀

Embedded C: The core DSLs use a textual notation with some tabular enhancements, for example, for decision tables (Fig. 2.45). However, as MPS' capability for handling graphical notations will improve, we will represent state machines as diagrams. ◀


```
module DecisionTableExample from cdesignpaper.gswitch imports nothing {

  enum mode { MANUAL; AUTO; FAIL; }

  mode nextMode(mode mode, int8_t speed) {
    return mode, FAIL
```

	mode == MANUAL	mode == AUTO
speed < 30	MANUAL	AUTO
speed > 30	MANUAL	MANUAL

```
;
  } nextMode (function)
}
```

Figure 2.45: Decision tables use a tabular notation. It is embedded seemlessly into a C program.

Selection of a concrete syntax is simple for domain user DSLs if there is an established notation in the domain. The challenge then is to replicate this notation as closely as possible with the DSL, while cleaning up possible inconsistencies in the notation (since presumably it had not been used formally before). I like to use the term "strongly typed word" in this case[78].

For DSLs targeted at developers, a textual notation is usually a good starting point, since developers are used to working with text, and they are very productive with it. Tree views, and specific visualizations are often useful, to present outlines, hierarchies or overviews, but not necessarily for editing. Textual notations also integrate well with existing development infrastructures.

[78] In some cases it is useful to come up with a better notation than the one used historically. This is especially true if the historic notation is Excel.)

Embedded C: C is the baseline for embedded systems, and everybody is familiar with it. A textual notation is useful for many concerns in embedded systems. Note that several languages create visualizations on the fly, for example for module dependencies, component dependencies and component instance wirings. The graphviz tool is used here since it provides decent auto-layout. ◀

There are very few DSLs where a *purely* graphical notation makes sense because in most cases, some textual languages are embedded in the diagrams or tables: state machines have expressions embedded the guards and statements in the actions (Fig. 2.48); component diagrams use text for specifications of operations in interfaces, maybe using expressions for preconditions; block diagrams use a textual syntax for the implementation/parametrization of the blocks (Fig. 2.46); tables may embed textual notations in the cells (Fig. 2.47). Integrating textual languages into graphical ones is becoming more and more important, and tool support is improving.

A text box where textual code can be entered without language support should only be used as a last resort. Instead, a textual notation, with additional graphical visualizations should be used

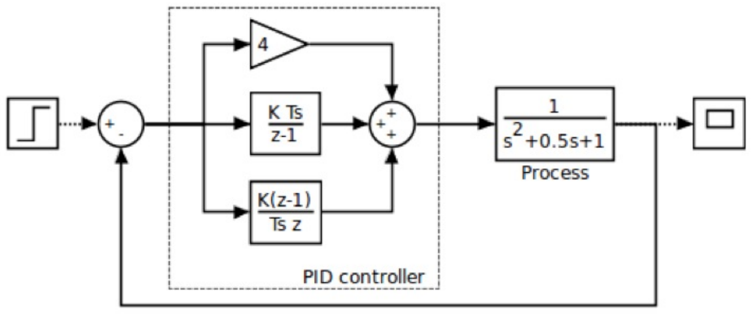


Figure 2.46: A block diagrams built with the Yakindu modeling tools. A textual DSL is used to implement the behavior in the blocks. While the textual DSL is not technically integrated with the graphical notation (separate viewponts), semantic integration is provided.

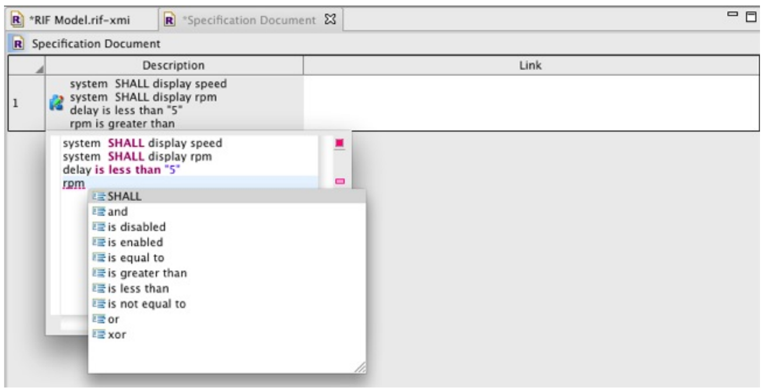
Note that initially, domain users prefer a graphical notation, because of the perception that things that are described graphically are simple(r) to comprehend. However, what is most important regarding comprehensibility is the alignment of the domain concepts with the abstractions in the language. A well-designed textual notation can go a long way. Also, textual languages are more productive once the learning curve has been overcome. I have had several cases where domain users started preferring textual notations later in the process.

In my consulting practice, I almost always start with a textual notation and try to stabilize language abstractions. Only then will I engage in a discussion about whether a graphical notations on top of the textual one is necessary. Often it is not, and if it is, we have avoided iterating the implementation of the graphical editor implementation, which, depending on the tooling, can be a lot of work.

■ *IDE Supportability* For textual languages, it is important to keep in mind if and how a syntax can be support by the IDE, especially regarding code completion. Consider query languages. An example SQL query looks like this:

```
SELECT field1, field2 FROM aTable WHERE ...
```

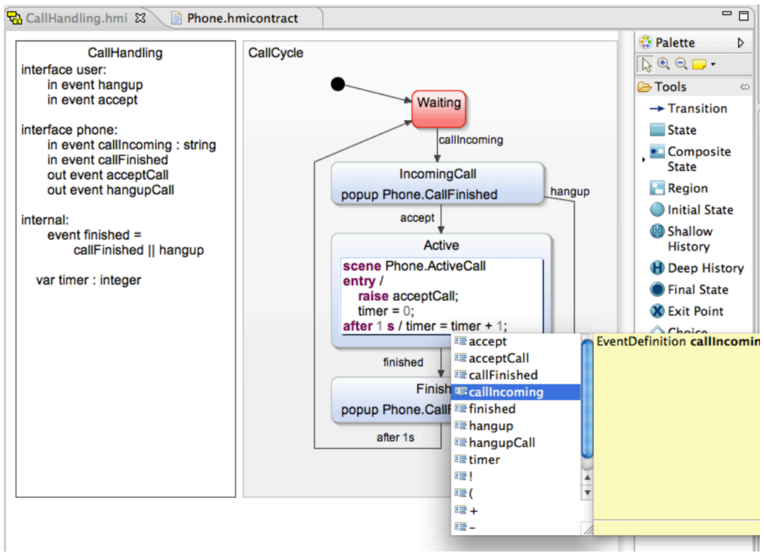
When entering this query the IDE cannot provide code completion for the fields after the `SELECT` because at this point the table has not yet



	Description	Link
1	system SHALL display speed system SHALL display rpm delay is less than "5" rpm is greater than	

```
system SHALL display speed
system SHALL display rpm
delay is less than "5"
rpm
```

Figure 2.47: The Yakindu Requirements tools integrates a textual DSL for formal requirements specification into a table view. The textual specifications are stored as text in the requirements database; consequently, the entities defined textually cannot be referenced (which is not a problem in this domain).



```
CallHandling
interface user:
    in event hangup
    in event accept

interface phone:
    in event callIncoming : string
    in event callFinished
    out event acceptCall
    out event hangupCall

internal:
    event finished =
        callFinished || hangup

  var timer : integer
```

```
scene Phone.ActiveCall
entry /
   raise acceptCall;
   timer = 0;
after 1 s / timer = timer + 1;
```

Figure 2.48: The Yakindu State Chart Tools support the use of Xtext DSLs in actions and guard conditions of state machines, mixing textual and graphical notations. The DSL can even be exchanged, to support domain specific action languages, for example for integrating with user interface specifications. In this case, the textual specification are stored as the AST in terms of EMF, not as text.

specified. A more suitable syntax, with respect to IDE support, would be

```
FROM aTable SELECT field1, field2 WHERE ...
```

because now the IDE can provide support code completion for the fields based on the table name that has already been entered when you specify the fields.

SQL is a relatively old language and IDE concerns where probably not very important at the time. More modern query language such as HQL or Linq in fact use the more IDE friendly syntax.

Note that tool supportability in general is not fundamentally different in graphical and textual languages. While IDEs for textual languages can provide code completion, the palette or the context buttons in a graphical DSL play the same role. I often hear that a graphical DSL is more suitable for simulation (because the execution of the program can be animated on the graphical notation). However, this is only true if the graphical notation works well in the first place. A textual program can also be animated; a debugger essentially does just that.

■ *Relationship to Hierarchical Domains* Domains at low D are most likely best expressed with a textual or symbolic concrete syntax. Obvious examples include programming languages at D_0. Mathematical expressions, which are also very dense and algorithmic, use a symbolic notation. As we progress to higher Ds, the concepts become more and more abstract, and as state machines and block diagrams illustrate, graphical notations become useful. However, these two notations are also a good example of language embedding since both of them require expressions: state machines in guards and actions (Fig. 2.48), and block diagrams as the implementation of blocks (Fig. 2.46 and Fig. 3.5). Reusable expression languages should be embedded into the graphical notations. In case this is not supported by the tool, viewpoints may be an option. One viewpoint could use a graphical notation to define coarse-grained structures, and a second viewpoint uses a textual notation to provide "implementation details" for the structures defined by the graphical viewpoint[79].

[79] Not every tool can support every (combination of) form of concrete syntax, so this aspect is limited by the tool, or drives tool selection.

Embedded C: As the graphical notation for state machines will become available, the C expression language that is used in guard conditions for transitions will be usable as labels on the transition arrows. In the table notation for state machines, C expressions can be embedded in the cells as well. ◀

3
Fundamental Paradigms

Every DSL is different. It is driven by the domain for which it is built. However, as it turns out, there are also a number of commonalities between DSLs. These can be handled by modularizing and reusing (parts of) DSLs as discussed in the last section of the previous chapter. In this *section we look at common paradigms for describing DSL structure and behavior.*

3.1 Structure

Languages have to provide means of structuring large programs in order to keep them manageable. Such means include modularization and encapsulation, specification vs. implementation, specialization, types and instances as well as partitioning.

The language design alternatives described in this section are usually not driven directly by the domain, or the domain experts guiding the design of the language. Rather, they are often brought in by the language designer as a means of managing overall complexity. For this reason they may be hard to "sell" to domain experts.

3.1.1 Modularization and Visibility

DSLs often provide some kind of logical unit structure, such as namespaces or modules. Visibility of symbols may be restricted to the same unit, or to referencing ("importing") units. Symbols may be declared as public or private, the latter making them changeable without consequences for using modules. Some form of namespaces and visibility is necessary in almost any DSL. Often there are domain concepts that can play the role of the module, possibly oriented towards the structure of the organization in which the DSL is used.

Most contemporary programming languages use some form of namespaces and visibility restriction as their top level structure.

Embedded C: As a fundamental extension to C, this DSL contains modules with visibility specifications and imports. Functions, state machines, tasks and all other top-level concepts reside in modules. Header files (which are effectively a poor way of managing symbol visibility) are only used in the generated low level code and not relevant to the user of mbeddr C. ◀

Component Architecture: Components and interfaces live in namespaces. Components are implementation units, and are always private. Interfaces and data types may be public or private. Namespaces can import each other, making the public elements of the imported namespace visible to the importing namespace. The OSGi generator creates two different bundles: an interface bundle that contains the public artifacts, and an implementation bundle with the components. In case of a distributed system, only the interface bundle is deployed on the client. ◀

Pension Plans: Pension plans constitute namespaces. They are grouped into more coarse-grained packages that are aligned with the structure of the pension insurance business. ◀

3.1.2 *Partitioning*

Partitioning refers to the breaking down of programs into several physical units such as files (typically each model fragment is stored in its own partition). These physical units do not have to correspond to the logical modularization of the models within the partitions. For example, in Java a public class has to live in a file of the same name (logical module == physical partition), whereas in C# there is no relationship between namespace, class names and the physical file and directory structure. A similar relationship exists between partitions and viewpoints, although in most cases, different viewpoints are stored in different partitions.

If a repository-based tool is used, the importance of paritioning is greatly reduced. Although even in that case, there may be a set of federated and distributed repositories that can be considered partitions.

Note that a reference to an element should not take into account the partition in which the target element lives. Instead, it should only use the logical structure. Consider an element `E` that lives in a namespace `x.y`, stored in a partition `mainmodel`. A reference to that element should be expressed as `x.y.E`, not as `mainmodel.E` or `mainmodel/x.y.E`. This is important to allow elements to move freely between partitions without this leading to updates of all references to the element.

Partitioning may have consequences for language design. Consider a textual DSL where an concept A contains a list of instances of concept B. The B instances then have to be physically nested within an instance of A in the concrete syntax. If there are many instances of B in a given model, they cannot be split into several files, so these files may become big and result in performance problems. If such a split should be possible, this has to be designed into the language.

Component Architecture: A variant of this DSL that was used in another project had to be changed to allow a namespaces to be spread over several files for reasons of scalability and version-control granularity. In the initial version, namespaces actually *contained* the components and interfaces. In the revised version, components and interfaces were owned by no other element, but model files (partitions) had a namespace declaration at the top, logically putting all the contained interfaces and components into this namespace. Since there was no technical containment relationship between namespaces and its elements, several files could now declare the same namespace. Changing this design decision lead to a significant reimplementation effort because all kinds of naming and scoping strategies changed. ◀

Other concerns influence the design of a partitioning strategy as well:

Change Impact Which partition changes as a consequence of a particular change of the model (changing an element name might require changes to all references to that element from other partitions)

Link Storage Where are links stored (are they always stored in the model that logically "points to" another one)?, and if not, how/where/when to control reference/link storage.

Model Organization Partitions may be used as a way of organizing the overall model. This is particularly important if the tool does not provide a good means of presenting the overall logical structure of models and finding elements by name and type. Organizing files with meaningful names in directory structures is a workable alternative.

Tool Chain Integration Integration with existing, file based tool chains. Files may be the unit of check in/check out, versioning, branching or permission checking.

Another driver for using partitions is the scalability of the DSL tool. Beyond a certain file size, the editor may become sluggish.

It is often useful to ensure that each partition is processable separately to reduce processing times. An alternative approach supports the explicit definition of those partitions that should be processed in a given processor run (or at least a search path, a set of directories, to find the partitions, like an include path in C compilers or the Java classpath). You might even consider a separate build step to combine the results created from the separate processing steps of the various partitions (again like a C compiler: it compiles every file separately into an object file, and then the linker handles overall symbol/reference resolution and binding).

The partitioning scheme may also influence users' team collaboration when editing models. There are two major collaboration models: real-time and commit-based. In real-time collaboration, a user sees his model change in real time as another user changes that same model. Change propagation is immediate. A database-backed repository is often a good choice regarding storage, since the granularity tracked by the repository is the model element. In this case, the partitioning may not be visible to the end user, since they just work "on the repository". This approach is often (at least initially) preferred by non-programmer DSL users. The other collaboration mode is commit-based where a user's changes only make it to the repository if he performs a *commit*, and incoming changes are only visible after a user has performed an *update*. While this approach can be used with database-backed repositories, it is most often used with file-based storage. In this case, the partitioning scheme is visible to DSL users, because it is those files they commit or update. This approach tends to be preferred by developers, maybe because well-known versioning tools have used the approach for a long time.

3.1.3 Specification vs. Implementation

Separating specification and implementation supports plugging in different implementations for the same specification and hence provides a way to "decouple the outside from the inside"[1]. This supports the exchange of several implementations behind a single interface. This is often required as a consequence of the development process: one stakeholder defines the specification and a client, whereas another stakeholder provides one or more implementations.

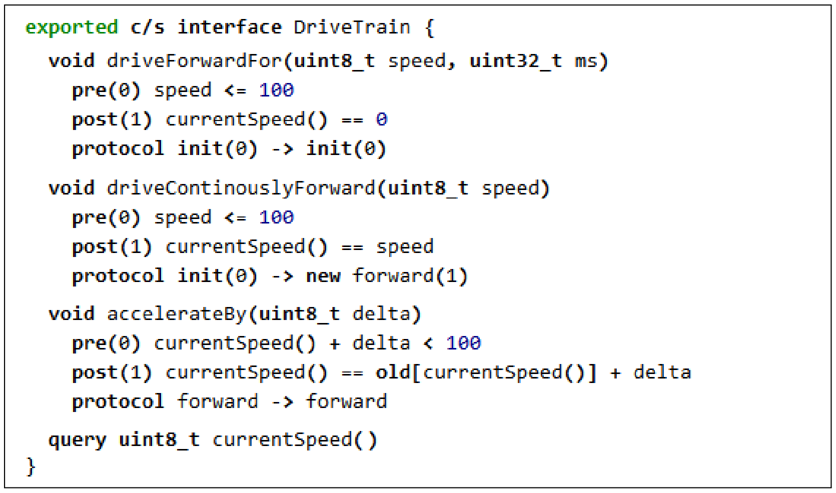
[1] Interfaces, pure abstract classes, traits or function signatures are a realization of this concept in programming languages.

A challenge for this approach is how to ensure that all implementations are consistent with the specification. Traditionally, only the structural/syntactic/signature compatibility is checked. To ensure semantic compatibility, additional means that specify the expected *behavior* are required. This can be achieved with pre or post conditions, invariants or protocol state machines.

The separation of specification and implementation can also have positive effects on scalability and performance. If the specification and implementation are separated into different fragments, then, in order to type check a client's access to some provided service, only the fragment that contains the specification has to be loaded/parsed/checked, which is obviously faster than processing complete implementation.

Embedded C: This DSL adds interfaces and components to C. Components provide or use one or more interfaces. Different components can be plugged in behind the same interface. To support semantic specifications, the interfaces support pre- and post conditions as well as protocol state machines. Fig. 3.1 shows an example. Although these specifications are attached to interfaces, they are actually checked (at runtime) for all components that provide the respective interface. ◀

Refrigerators: Cooling programs can refer to entities defined as part of the refrigerator hardware as a means of accessing hardware elements (compressors, fans, valves). To enable cooling programs to run with different, but similar hardware configurations, the hardware structure can use "trait inheritance", where a hard-



```
exported c/s interface DriveTrain {
  void driveForwardFor(uint8_t speed, uint32_t ms)
    pre(0) speed <= 100
    post(1) currentSpeed() == 0
    protocol init(0) -> init(0)
  void driveContinouslyForward(uint8_t speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init(0) -> new forward(1)
  void accelerateBy(uint8_t delta)
    pre(0) currentSpeed() + delta < 100
    post(1) currentSpeed() == old[currentSpeed()] + delta
    protocol forward -> forward
  query uint8_t currentSpeed()
}
```

Figure 3.1: An interface using semantic specifications. Preconditions check the values of arguments for validity. Postconditions express constraints on the values of `query` operations after the execution of the operation. Notice how the value of the `query` before executing the operation can be referred to (the `old` keyword used in the postcondition for `accelerateBy`). In addition, protocols constrain the valid sequence of operation invocations. For example, the `accelerateBy` operation can only be used if the protocol state machine is already in the `forward` state. The system gets into the `forward` state by invoking the `driveContinuoslyForward` operation.

ware trait defines a set of hardware elements, acting as a kind of interface. Other hardware configurations can inherit these traits. As long as cooling programs are only written against traits, they work with any refrigerator that implements the particular set of traits against which the program is written. ◀

3.1.4 *Specialization*

Specialization enables one entity to be a more specific variant of another one. Typically, the more specific one can be used in all contexts where the more general one is expected (the Liskov substitution principle[2]). The more general one may be incomplete, requiring the specialized ones to "fill in the holes". Specialization in the context of DSLs can be used for implementing variants or for evolving a program over time.

[2]

In GPLs, we know this approach from class inheritance. "Leaving holes" is realized by abstract methods.

Note that defining the semantics of inheritance for domain-specific language concepts is not always easy. The various approaches found in programming languages, as well as the fact that some of them lead to problems (multiple inheritance, diamond inheritance, linearization, or code duplication in Java's interface inheritance) shows that this is not a trivial topic. It is a good idea to just copy a suitable approach *completely* from a programming language where inheritance seems to work well. Even small changes can make the whole approach inconsistent.

Pension Plans: The customer using this DSL had the challenge of creating a huge set of pension plans, implementing changes in relevant law over time, or implementing related plans for different customer groups. Copying complete plans and then making adaptations was not feasible, because this resulted in a maintenance nightmare: a large number of similar, but not identical pension plans. Hence the DSL provides a way for pension plans to inherit from one another. Calculation rules can be marked *abstract* (requiring to be overwritten in sub-plans), *final* rules are not overwritable. Visibility modifiers control which rules are considered "implementation details". ◀

Refrigerators: A similar approach is used in the refrigerator DSL. Cooling programs can specialize other cooling programs. Since the programs are fundamentally state-based, we had to define what exactly it means to override a cooling program: a subprogram can add additional event handlers and transitions to states. New states can also be added. However, states defined in the super-program cannot be removed. ◀

3.1.5 *Types and Instances*

Types and instances refers to the ability to define a structure that can be parametrized when it is instantiated. It supports reusing the fixed parts, and expressing variability via the parameters.

In programming languages we know this from classes and objects (where constructor parameters are used for parametrization) or from components (where different instances can be connected differently to other instances).

Embedded C: Apart from C's `structs` (which are instantiatable data structures) and components (which can be instantiated and connected), state machines can be instantiated as well. Each instance can be in a different state at any given time. ◀

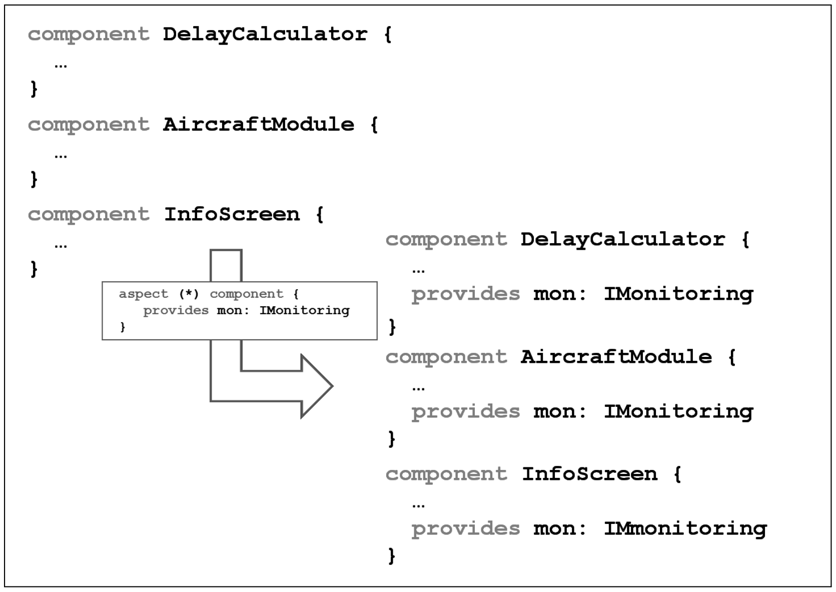
3.1.6 *Superposition and Aspects*

Superposition refers to the ability to merge several model fragments according to some DSL-specific merge operator. Aspects provide a way of "pointing to" several locations in a program based on a pointcut operator (essentially a query over a program or its execution), adapting the model in ways specified by the aspect. Both approaches support the compositional creation of many different model variants from the same set of model fragments.

This is especially important in the context of product line engineering and is discussed in Section **??**.

Component Architecture: This DSL provides a way of advising component definitions from an aspect (Fig. 3.2). An aspect may introduce an additional port `provided port mon: IMonitoring` that allows a central monitoring component to query the adviced components via the `IMonitoring` interface. ◀

WebDSL: Entity declarations can be *extended* in separate modules. This makes it possible to declare in one module all data model declarations of a particular feature. For example, in the *researchr* application, a `Publication` can be `Tag`ged, which requires an extension of the `Publication` entity. This extension is defined in the `tag` module, together with the definition of the `Tag` entity. This is essentially a use of superposition. ◀



```
component DelayCalculator {
  ...
}
component AircraftModule {
  ...
}
component InfoScreen {
  ...
}
```

```
aspect (*) component {
   provides mon: IMonitoring
}
```

```
component DelayCalculator {
  ...
  provides mon: IMonitoring
}
component AircraftModule {
  ...
  provides mon: IMonitoring
}
component InfoScreen {
  ...
  provides mon: IMmonitoring
}
```

Figure 3.2: The aspect component contributes an additional required port to each of the other components defined in the system.

3.1.7 *Versioning*

Often, variability over time of elements in DSL programs have to be tracked. One alternative is to simply version the model files using existing version control systems, or the version control mechanism built into the language workbench. However, this requires users to interact with often complex version control systems and prevents domain-specific adaptations of the version control strategy.

The other alternative is to make versioning and tracking over time a part of the language. For example, model elements may be tagged with version numbers or specify a revision chain by pointing to a previous revision, enforcing compatibility constraints between those revisions. Instead of declaring explicit versions, business data is often time-dependent, where different revisions of a business rule apply to different periods of time. Support for these approaches can be built directly into the DSL, with various levels of tool support.

Embedded C: No versioning is defined into the DSL. Users work with MPS' integration with popular version control systems. Since this DSL is intended for use by programmers, working with existing version control systems is not a problem. ◀

Component Architecture: Components can specify a `new version of` reference to another component In this case, the new version may specify additional provided ports with the same interfaces or new versions of these interfaces. It may deprecate required ports. Effectively, this means that the new version of something must be replacement-compatible with the old version (the Liskov substitution principle again). ◀

Pension Plans: In the pension workbench, calculation rules declare applicability periods. This supports the evolution of calculation rules over time, while retaining reproducability for calculations performed at an earlier point in time. Since the Intentional Domain Workbench is a projectional tool, pension plans can be shown with only the version of a rule valid for a given point in time. ◀

3.2 *Behavior*

The behavior expressed with a DSL must of course be aligned with the needs of the domain. However, in many cases, the behavior required for a domain can be derived from well-known behavioral paradigms[3], with slight adaptations or enhancements, or simply interacting with domain-specific structures or data.

[3] The term *Model of Computation* is also used to refer to behavioral paradigms. I prefer behavioral paradigm because the term *Model* is obviously heavily overloaded in the DSL/MDSD space already.

Note that there are two kinds of DSLs that don't make use of these kinds of behavior descriptions. Some DSLs really just specify structures. Examples include data definition languages or component description languages (although both of them often use expressions for derived data, data validation or pre- and post-conditions). Other DSLs specify a set of expectations regarding some behavior (declaratively), and the generator creates the algorithmic implementation. For example, a DSL may specify, simply with a tag such as *async*, that the communication between two components shall be asynchronous. The generator then maps this to an implementation that behaves according to this specification.

Component Architecture: The component architecture DSL is an example of a structure-only DSL, since it only describes black box components and their interfaces and relationships. It uses the specification-only approach to specify whether a component port is intended for synchronous or asynchronous communication. ◀

Embedded C: The component extension to provides a similar notion of interfaces, ports and components as in the previous example. However, since here they are directly integrated with C, C expression can be used for pre- and post-conditions of interface operations (see Fig. 3.1). ◀

Using an established behavioral paradigm for a DSL has several advantages[4]. First, it is not necessarily simple to define consistent and correct semantics in the first place. By reusing an existing paradigm,

[4] Which is why we discuss these paradigms in this book.

one can learn about advantages and drawbacks from existing experience. Second, a paradigm may already come with existing means for performing interesting analyses (as in model checking or SMT solving) that can easily be used to analyse DSL programs. Third, there may be existing generators from a behavioral paradigm to an efficient executable for a given platform (state machines are a prime candidate). By generating a model in a formalism for which such a generator exists, we reduce the effort for building an end-to-end generator. If our DSL uses the same behavioral paradigm as the language for which the genertor exists, writing the necessary transformation is straight forward (from a semantic point of view).

The last point emphasizes that using an existing paradigm for a DSL (e.g. state-based) does not mean that the concepts have to directly use the abstractions used by that paradigm (just because a program is state-based, does not mean that the concept that acts as a state has to be called state, etc.).

This is only an overview over a few paradigms; many more exist. I refer to the excellent Wikipedia entry on *Programming Paradigms* and to the book

This section describes some of the most well-known behavioral paradigms that can serve as useful starting points for behavior descriptions in DSLs. In addition to describing the paradigm, we also briefly investigate how easy programs using the paradigm can be analyzed, and how complicated it is to build debuggers.

3.2.1 *Imperative*

For many people, often including domain experts, this approach is easy to understand. Hence it is often a good starting point for DSLs.

Imperative programs consist of a sequence of statements, or instructions, that change the state of the program. This state may be local to some kind of module (e.g., a procedure or an object), global (as in global variables) or external (when communicating with peripheral devices). Procedural and object-oriented programming are both imperative, using different means for structuring and (in case of OO) specialization. Because of aliasing and side effects, imperative programs are expensive to analyse. Debugging imperative programs is straight forward and involves stepping through the instructions and watching the state change.

Embedded C: Since C is used as a base language, this language is fundamentally imperative. Some of the DSLs on top of it use other paradigms (the state machine extension is state-based, for example). ◀

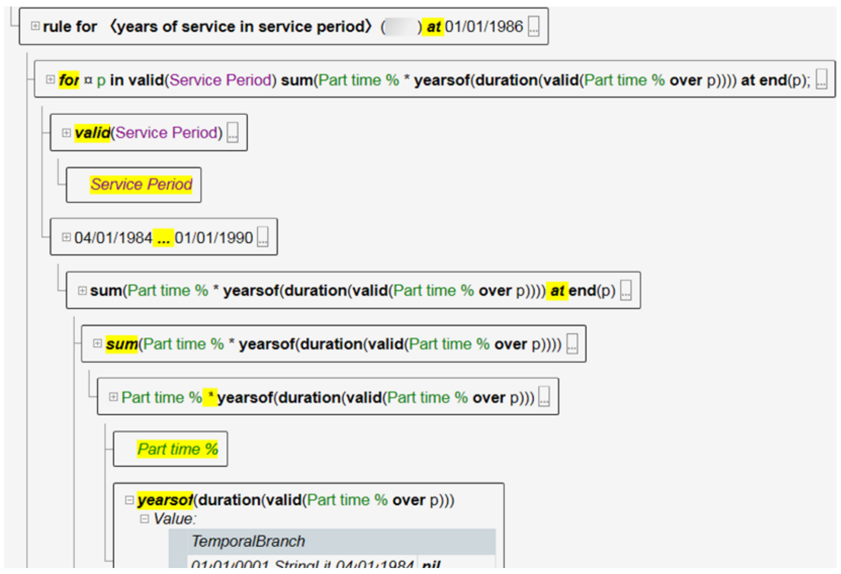
Refrigerators: The cooling language integrates various paradigms, but contains sequences of statements to implement aspects of the overall cooling behavior. ◀

3.2.2 *Functional*

Functional programming uses functions as the core abstraction. In purely functional programming, a function's return value only depends on the values of its arguments. Calling the same function several times with the same argument values returns the same result (that value may even be cached!). Functions cannot access global mutable state, no side effects are allowed. These characteristics make functional programs very easy to analyze and optimize. These same characteristics, however, also make purely functional programming relatively useless, because it cannot affect its environment (after all, this would be a side effect). So, functional programming is often only used for parts ("calculation core") of an overall program and integrates with, for example, an imperative part that deals with IO.

Since there is no state to watch change as the program steps through instructions, debugging can be done by simply showing all intermediate results of all function calls as some kind of tree, basically "inspecting" the state of the calculation. This makes building debuggers relatively simple.

Pension Plans: The calculation core of pension rules is functional. Consequently, a debugger has been implemented that, for a given set of input data, shows the rules as a tree where all intermediate results of each function call are shown (Fig. 3.3). No "step through" debugger is necessary. ◄



```
⊞ rule for 〈years of service in service period〉 (    ) at 01/01/1986
  ⊞ for ¤ p in valid(Service Period) sum(Part time % * yearsof(duration(valid(Part time % over p)))) at end(p);
    ⊞ valid(Service Period)
      Service Period
    ⊞ 04/01/1984 ... 01/01/1990
      ⊞ sum(Part time % * yearsof(duration(valid(Part time % over p)))) at end(p)
        ⊞ sum(Part time % * yearsof(duration(valid(Part time % over p))))
          ⊞ Part time % * yearsof(duration(valid(Part time % over p)))
            Part time %
            ⊟ yearsof(duration(valid(Part time % over p)))
              ⊟ Value:
                  TemporalBranch
                  01/01/0001 String 04/01/1984 nil
```

Figure 3.3: Debugging functional programs can be done by showing the state of the calculation, for example as a tree.

An important subset of functional programming is pure expressions (as in `i > 3*2 + 7`. Instead of calling functions, operators are used. However, operators are just infix notations for function calls. Usually the operators are hard wired into the language and it is not possible for users to define their own functional abstractions. The latter the main differentiator to functional programming in general. It also limits expressivity, since it is not possible to modularize an expression or to reuse expressions by packaging into a user-defined function. Consequently, only relatively simply tasks can be addressed with a pure expression language[5].

[5] However, many DSLs do not require anything more sophisticated, especially if powerful domain-specific operators are available. So, while expression languages are limited in some sense, they are still extremely useful and widespread.

Embedded C: We use expressions in the guard conditions of the state machine extension as well as in pre- and post-conditions for interface operations. In both cases it is not possible to define or call external functions. Of course, (a subset of) C's expression language is reused here. ◀

3.2.3 *Declarative*

Declarative programming can be considered the opposite of imperative programming (and, to some extent, functional programming). A declarative program does not specify any control flow, it does not specify a sequence of steps of a calculation. A declarative program only specifies *what* the program should accomplish, not *how*. This is often achieved by specifying a set of properties, equations, relationships or constraints. Some kind of evaluation engine then tries to find solutions. The particular advantage of this approach is that it is not predefined how a solution is found, the evaluation engine has a lot of freedom in doing so, possibly using different approaches in different environments, or evolving the approach over time[6]. This large degree of freedom often makes finding the solution expensive – trial and error, backtracking or exhaustive search may be used[7]. Debugging declarative programs can be hard since the solution algorithm may be very complex and possibly not even be known to the user of the language.

[6] For example, the strategies for implementing SAT solvers have evolved quite a bit over time. SAT solvers are much more scalable today. However, the formalism for describing the logic formulas that are processed by SAT solvers have not changed

[7] So, often users have to provide hints to the engine to make it run fast enough or scale to programs of relevant size. In practice, declarative programming is often not as "pure" as it is in theory.

Declarative programming has many important subgroups and use cases. For *concurrent programs*, a declarative approach allows the efficient execution of a single program on different parallel hardware structures. The compiler or runtime system can allocate the program to available computational resources. In *constraint programming*, the programmer specifies constraints between a set of variables. The engine tries to find values for these variables that satisfy all constraints. Solving mathematical equation systems is an example, as is solving sets of Boolean logic formulas. *Logic programming* is another subparadigm of declarative programming where users specify logic clauses (facts and relations) as well as queries. A theorem prover tries to solve the queries.

The Prolog language works this way

Component Architecture: This DSL specifies timing and resource characteristics for component and interface operations. Based on this data, one could run an algorithm which allocates the component instances to computing hardware so that the hardware is used as efficiently as possible, while at the same time reducing the amount of network traffic. This is an example of constraint solving used to synthesize a schedule. ◀

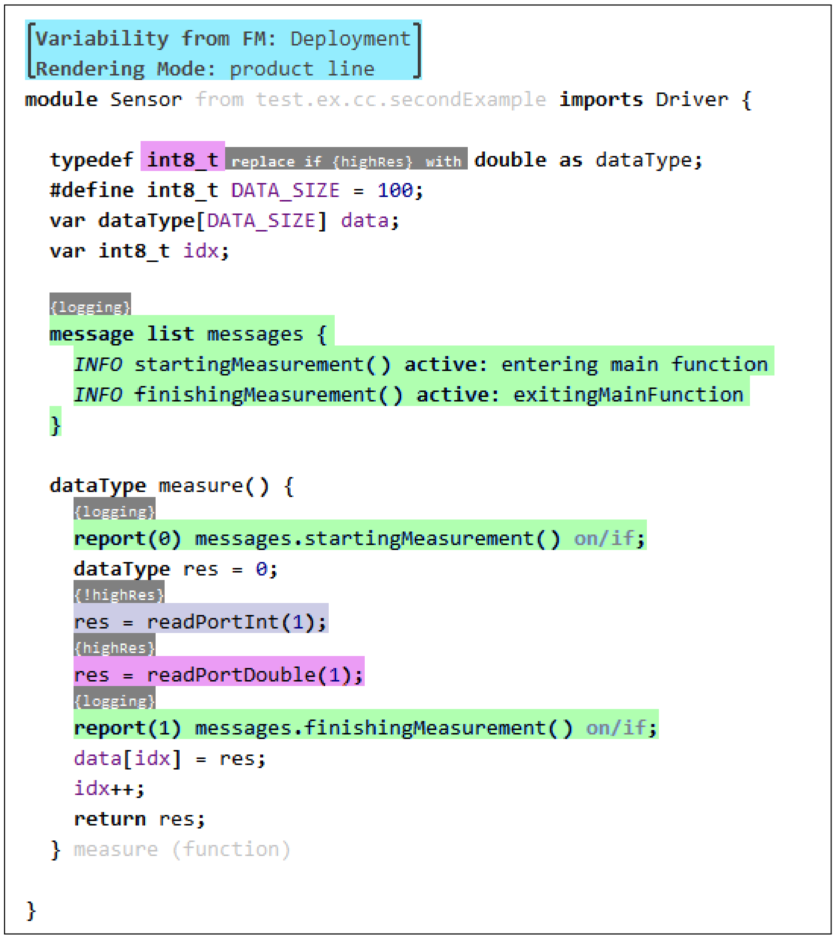
Embedded C: This DSL supports presence conditions for product line engineering. A presence condition is a Boolean expression over a set of configuration features that determines whether the associated piece of code is present for a given combination of feature selections (Fig. 3.4). To verify the structural integrity of programs in the face of varying feature combinations, constraint programming is used (to ensure that there is no configuration of the program where a reference to a symbol is included, but the referenced symbol is not). From the program, the presence conditions and the feature model a set of Boolean equations is generated. A solver then makes sure they are consistent by trying to find an example solution that violates the Boolean equations. ◀

Example: The Yakindu DAMOS block diagram editor supports custom block implementation based on the Mscript language. It (Section 3.5) supports declarative specification of equations between input and output parameters of a block. A solver comes up with an closed, sequential solution that efficiently calculates the output of an overall block diagram. ◀

Example: Another example for declarative programming is the type system DSL used by MPS itself. Language developers specify a set of type equations containing free type variables, among other things. A unification engine tries to solve the set of equations by assigning actual types to the free type variables so that the set of equations is consistent. We describe this approach in detail inSection **??**. ◀

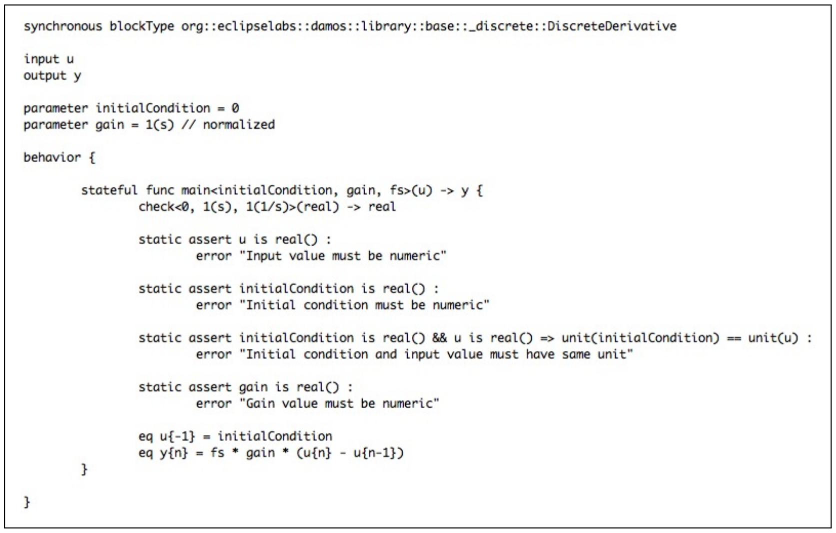
3.2.4 *Reactive/Event-based/Agent*

In this paradigm, behavior is triggered based on events received by some entity. Events may be created by another entity or by the environment (through a device driver). Reactions are expressed by the creation of other events. Events may be globally visible or explicitly routed between entities, possibly using filters and/or using priority queues. This approach is often used in embedded systems that have to interact with the real world, where the real world produces events as it changes. A variant of this approach queries input signals at inter-



```
[Variability from FM: Deployment]
[Rendering Mode: product line   ]
module Sensor from test.ex.cc.secondExample imports Driver {

  typedef int8_t replace if {highRes} with double as dataType;
  #define int8_t DATA_SIZE = 100;
  var dataType[DATA_SIZE] data;
  var int8_t idx;

  {logging}
  message list messages {
    INFO startingMeasurement() active: entering main function
    INFO finishingMeasurement() active: exitingMainFunction
  }

  dataType measure() {
    {logging}
    report(0) messages.startingMeasurement() on/if;
    dataType res = 0;
    {!highRes}
    res = readPortInt(1);
    {highRes}
    res = readPortDouble(1);
    {logging}
    report(1) messages.finishingMeasurement() on/if;
    data[idx] = res;
    idx++;
    return res;
  } measure (function)

}
```

Figure 3.4: This module contains variability expressed with presence conditions. The affected program elements are highlighted in a color that represents the condition. If the feature **highRes** is selected, the code uses a **double** instead of an **int8_t**. The log messages are only included if the **logging** feature is selected. Note that one cannot just depend on single features (such as **logging**) but also on arbitrary expressions such as **logging && highRes**.



```
synchronous blockType org::eclipselabs::damos::library::base::_discrete::DiscreteDerivative

input u
output y

parameter initialCondition = 0
parameter gain = 1(s) // normalized

behavior {

        stateful func main<initialCondition, gain, fs>(u) -> y {
                check<0, 1(s), 1(1/s)>(real) -> real

                static assert u is real() :
                        error "Input value must be numeric"

                static assert initialCondition is real() :
                        error "Initial condition must be numeric"

                static assert initialCondition is real() && u is real() => unit(initialCondition) == unit(u) :
                        error "Initial condition and input value must have same unit"

                static assert gain is real() :
                        error "Gain value must be numeric"

                eq u{-1} = initialCondition
                eq y{n} = fs * gain * (u{n} - u{n-1})
        }

}
```

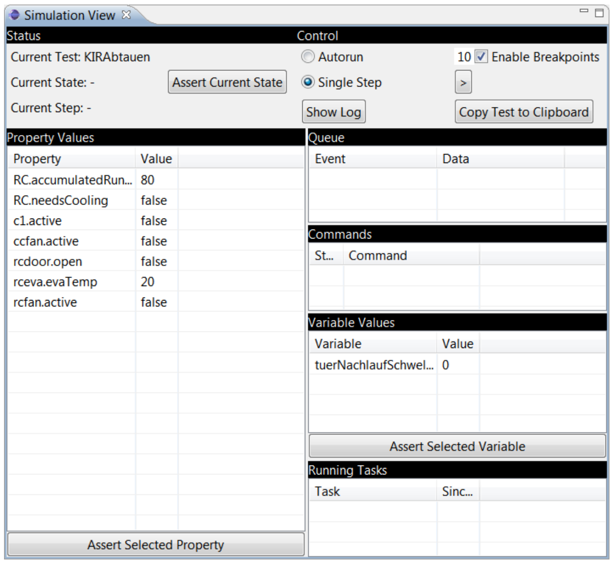
Figure 3.5: An Mscript block specifies input and output arguments of a block (u and v) as well as configuration parameters (*initialCondition* and *gain*). The assertions specify constraints on the data the blocks works with. The *eq* statements specify how the output values are calculated from the input values. Stateful behaviors are supported, where the value for the n-th step depends on values from previous steps (e.g., $n - 1$).

vals controlled by a scheduler and considers changes in input signals as the events.

Refrigerators: The cooling algorithms are reactive programs that control the cooling hardware based on environment events. Such events include the opening of a refrigerator door, the crossing of a temperature threshold, or a timeout that triggers defrosting of a cooling compartment. Events are queued, and the queues are processed in intervals determined by a scheduler. ◀

Debugging is simple if the timing/frequency of input events can be controlled. Visualizing incoming events and the code that is triggered as a reaction is relatively simple. If the timing of input events cannot be controlled, then debugging can be almost impossible, because humans are way too slow to fit "in between" events that may be generated by the environment in rapid succession. For this reason, various kinds of simulators are used to debug the behavior of reactive systems, and sophisticated diagnostics regarding event frequencies or queue filling levels may have to be integrated into the programs as they run in the real environment.

Refrigerators: The cooling language comes with a simulator (Fig. 3.6) based on an interpreter where the behavior of a cooling algorithm can be debugged. Events are explicitly created by the user, on a time scale that is compatible with the debugging process. ◀



Simulation View

Status

Current Test: KIRAbtauen

Current State: -

Current Step: -

Control

( ) Autorun
(●) Single Step

10 ☑ Enable Breakpoints

Property Values

Property	Value
RC.accumulatedRun...	80
RC.needsCooling	false
c1.active	false
ccfan.active	false
rcdoor.open	false
rceva.evaTemp	20
rcfan.active	false

Queue

Event	Data

Commands

St...	Command

Variable Values

Variable	Value
tuerNachlaufSchwel...	0

Running Tasks

Task	Sinc...

Figure 3.6: The simulator for the cooling language shows the state of the system (commands, event queue, value of hardware properties, variables and tasks). The program can be single-stepped. The user can change the value of variables or hardware properties as a means of interacting with the program.

3.2.5 *Dataflow*

The dataflow paradigm is centered around variables with dependencies (in terms of calculation rules) among them. As a variable changes, those variables that depend on the changing variable are recalculated.

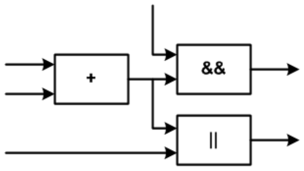


Figure 3.7: Graphical Notation for Flow

We know this approach mainly from two use cases. One is spreadsheets: cell formulas express dependencies to other cells. As the values in these other cells change, the dependent cells are updated. The other use case is data flow (or block) diagrams (Fig. 3.7, used in embedded software, extraction-transfer-load data processing systems and enterprise messaging/complex event processing. There, the calculations or transformations are encapsulated in the blocks, and the lines represent dependencies – the output of one blocks "flows" into input slot of another block. There are three different execution modes:

- The first one considers the data values as continuous signals. At the time one of the inputs changes, all dependent values are recalculated. The change triggers the recalculation, and the recalculation ripples through the dependency graph. This is the model used in spreadsheets.
- The second one considers the data values quantized, unique messages. Only if a message is available for all inputs, a new output message is calculated. The recalculation synchronizes on the availability of a message at each input, and upon recalculation, these messages are consumed. This approach is often used in ETL and CEP systems.
- The third approach is time triggered. Once again, the inputs are understood to be continuous signals, and a scheduler determines when a new calculation is performed. It also makes sure that the calculation "ripples through from left to right" in the correct order. This model is typically used in embedded systems.

Debugging these kinds of systems is relatively straight forward because the calculation is always in a distinct state. Dependencies and data flow, or the currently active block and the available messages can easily be visualized in a block diagram notation. Note that the calculation rules themselves are considered black boxes here, whose inside may be built from any other paradigm, often functional. Integrating debuggers for the inside of boxes is a more challenging task.

3.2.6 State-based

The state-based paradigm describes a system's behavior in terms of the states the system can be in, the transitions between these states as well as events that trigger these transitions and actions that are executed as states change. State machines are useful for systematically organizing the behavior of an entity. It can also be used to describe valid sequences of events, messages or procedure calls. State machines can be used in an event-driven mode where incoming events actually trigger transitions and the associated actions. Alternatively a state machine can be run in a timed mode, where a scheduler determines when event queues are checked and processed. Except for possible real-time issues, state machines are easy to debug by highlighting the contents of event queues and the current state[8].

[8] Except for the imperative paradigm and simple expression languages, state machines are probably the paradigm that is most often used in DSLs.

Embedded C: As mentioned before, this language provides an extension that supports directly working with state machines. Events can be passed into a state machine from regular C code or by mapping incoming messages in components to events in state machines that reside in components. Actions can contain arbitrary C code, unless the state machine shall be verifiable in which case actions may only create outgoing events or change statemachine-local variables. ◀

Refrigerators: The behavior of cooling programs is fundamentally state driven. A scheduler is used to execute the state machine in regular intervals. Transitions are triggered either by incoming, queued events or by changing property values of hardware building blocks. Note that this language is an example where a behavioral paradigm is used without significant alterations, but working with domain-specific data structures: refrigerator hardware and their properties. ◀

As mentioned before, state-based behavior description is also interesting in the context of model checking. The model checker either determines that the state chart conforms to a set of specifications or provides a counter example that violates the specifications. Specifications express something about sequences of states such as: "it is not possible that two traffic lights show green at the same time" or "whenever a pedestrian presses the `request` button, the pedestrian lights eventually will show green".

A good introduction to model checking can be found in .

In principle, any program can be represented as a state machine and can then model checked. However, creating state machines from, say, a procedural C program is non-trivial, and the state machines also become very big very quickly. State-based programs *are already* a state machine, and, they are typically not that big either (after all, they have to be understood by the developer who creates and maintains them). Consequently, many realistically-sized state machines can be model checked efficiently.

3.3 Combinations

The behavioral paradigm also plays a role in the context of language composition. If two to-be-composed languages use different behavioral paradigms, the composition can become really challenging. For example, combining a coninuous system (which works with continuous streams of data) with a discrete event-based system requires temporal integration. We won't discuss this topic in detail in this book[9]. However, it is obvious that combining systems that use the same paradigm is much simpler. Alternatively, some paradigms can be integrated relatively easily; for example, it is relatively simple to map a state-based system onto an imperative system.

[9] It is still very much a research topic

Many DSLs use combinations of various behavioral and structural paradigms described in this section[10]. A couple of combinations are very typical:

[10] Note how this observation leads to the desire to better modularize and reuse some of the above paradigms. Room for research :-)

- a data flow language often uses a functional, imperative or declarative language to describe the calculation rules that express the dependencies between the variables (the contents of the boxes in data flow diagrams or of cells in spreadsheets). Fig. 2.46 shows an example block diagram, and Fig. 3.5 shows an example implementation.

- state machines use expressions as transition guard conditions, as well as typically an imperative language for expressing the actions that are executed as a state is entered or left, or when a transition is executed. An example can be seen in Fig. 2.48

- reactive programming, where "black boxes" react to events, often use data flow or state-based programming to implement the behavior that determines the reactions.

- in purely structural languages, for example, those for expressing components and their dependencies, a functional/expression language is often used to express pre- and postconditions for operations. A state-based language is often used for protocol state machines, which determines the valid order of incoming events or operation calls.

Note that these combinations can be used to make well-established paradigms domain specific. For example, in the Yakindu State Chart Tools (Fig. 2.48), a custom DSL can be plugged into an existing, reusable state machine language and editor. One concrete example is an action language that references another DSL that describes UI structures. This way, the state machine can be used to orchestrate the behavior of the UI.

Some of the case studies used as examples in this part of the book also use combinations of several paradigms.

Pension Plans: The pension language uses functional abstractions with mathematical symbols for the core actuary mathematics. A functional language with a plain textual syntax is used for the higher-level pension calculation rules. A spreadsheet/data flow language is used for expressing unit tests for pension rules. Various nesting levels of namespaces are used to organize the rules, the most important of which is the pension plan. A plan contains calculation rules as well as test cases for those rules. Pension plans can specialize other plans as a means of expressing variants. Rules in a sub-plan can override rules in the plan from which the sub-plan inherits. Plans can be declared to be abstract, with abstract rules that have to be implemented in sub-plans. Rules are versioned over time, and the actual calculation formula is part of the version. Thus, a pension plan's behavior can be made to be different for different points in time. ◀

Refrigerators: The cooling behavior description is described as a reactive system. Events are produced by hardware elements (and their drivers). A state machine constitutes the top level structure. Within it, an imperative language is used. Programs can inherit from another program, overwriting states defined in the base program: new transitions can be added, and the existing transitions can be overridden as a way for an extended program to "plug into" the base program. ◀

4
Process Issues

Software development with DSLs requires a compatible development process. A lot of what's required is similar to what's required for working with any other reusable artifact such as a framework: a workable process must be established between those who build the reusable artifact and those who use it. Requirements have to flow in one direction, and a finished, stable, tested and document product has to be delived in the other direction. Also, using DSLs is a bit of a change for all involved, especially the domain experts. In this chapter we provide some guidelines regarding the process.

4.1 DSL Development

4.1.1 Requirements for the Language

How do you find out what your DSL should express? What are the relevant abstractions and notations? This is a non-trivial issue, in fact, it is one of the key issues in developing DSLs. It requires a lot of domain expertise, thought and iteration. The core problem is that you're trying to not just understand one problem, but rather a *class* of problems. Understanding and defining the extent and nature of this class of problems can be a lot of work. There are several typical ways of how to get started.

If you're building a technical DSL, the source for a language is often an existing framework, library, architecture or architectural pattern (inductive approach). The knowledge often already exists, and building the DSL is mainly about factoring the knowledge into a language: defining a notation, putting it into a formal language, and building generators to generate (parts of) the implementation code. In the process, you often also want to put in place reasonable defaults for some of the framework features, thereby increasing the level of abstraction and making framework use easier.

Embedded C: This was the approach taken by the extensible C case study. There is a lot of experience in embedded software development, and some of the most pressing challenges are the same throughout the industry. When the DSL was built, we talked to expert embedded software developers to find out what these central challenges were. We also used an inductive approach and looked at existing C code to indentify idioms and patterns. We then defined extensions to C that provided linguistic abstractions for the most important ones. ◀

In case of business domain DSLs, you can often mine the existing (tacit) knowledge of domain experts (deductive approach). In domains like insurance, science or logistics, domain experts are absolutely capable of precisely expressing domain knowledge. They do it all the time, often using Excel or Word. Similar to domain knowledge, other domain artifacts can also be exploited: for example, hardware structures or device features are good candidates for abstractions in the respective domains. So are existing user interfaces: they face users directly, and so are likely to contain core domain abstractions. Other sources are standards for an industry, or training material. Some domains even have an agreed upon ontology containing concepts relevant to that domain, and recognized as such by a community of stakeholders. DSLs can be (partly) derived from such domain ontologies.

Pension Plans: The company for which the pension DSL was built had a lot of experience with pension plans. This experience was mostly in the heads of (soon to be retiring) senior domain experts. They also already had the core of the DSL: a "rules language". The people who defined the pension plans would write rules into Word documents to "formally" describe the pension plan behavior. This was not terribly productive because of the missing tool support, but it meant that the core of the DSL was known. We still had to run a long series of workshops to figure out necessary changes to the language, clean up loose ends and discuss modularization and reuse in pension plans. ◀

In these two cases, it is pretty clear how the DSL is going to look like regarding core abstractions; discussions will be about details, notation, how to formalize things, viewpoints, partitioning and the like (note that those things can be pretty non-trivial, too!).

However, in the remaining third case, however, we are not so lucky. If no domain knowledge is easily available, we have to do an actual domain analysis, digging our way through requirements, stakeholder "war stories" and existing applications. People may be knowledgeable,

One of my most successful approaches in this case is to build straw men: trying to understand something, factor it into some kind of regular structure, and then re-explain that structure back to the stakeholders.

but they might be unable to conceptualize their domain in a structured way – it is then the job of the language designer to provide the structure and consistency that is necessary for defining a language. Co-evolving language and concepts (see below) is a successful technique especially in this case.

> **Refrigerators:** At the beginning of the project, all cooling algorithms were implemented in C. Specifications were written as prose (with tables and some physical formulas) in Word documents. It was not really clear at the beginning what the right abstraction level would be for a DSL suitable for the thermodynamics experts. It took several iterations to settle on the asynchronous, state-based structure described earlier. ◀

For your first DSL, try to catch case one or two. Ideally, start with case one, since the people who build the DSLs and supporting tools are often the same ones as the domain experts – software architects and developers.

4.1.2 *Iterative Development*

Some people use DSLs as an excuse to reintroduce waterfall processes. They spend months and months developing languages, tools, and frameworks. Needless to say, this is not a very successful approach. You need to iterate when developing the language.

Start by developing some deep understanding of a small part of the domain for which you build the DSL. Then build a little bit of language, build a little bit of generator and develop a small example model to verify what you just did. Ideally, implement all aspects of the language and processor for each new domain requirement before focusing on new requirements[1].

[1] IDE polishing is probably something you want to postpone a little bit, and not do it as part of every iteration

Especially newbies to DSLs tend to get languages and meta models wrong because they are not used to "think meta". You can avoid this pitfall by immediately trying out your new language feature by building an example model and developing a compatible generator to verify that you can actually generate the relevant artifacts.

> **Refrigerators:** In order to solidify our choices regarding language abstractions, we prototypically implemented several example refrigerators. During this process we found the need for more and more language abstractions. We noticed early that we needed a way to test the example programs, so we implemented the interpreter and simulator relatively early. In each iteration, we extended the language as well as the interpreter, so the domain experts could experiment with the language even though we did not have a C code generator yet. ◀

It is important that the language approaches some kind of stable state over time (Fig. 4.1). As you iterate, you will encounter the following situation: domain experts express requirements that may sound inconsistent. You add all kinds of exceptions and corner cases to the language. You language grows in size and complexity. After a number of these exceptions and corner cases, ideally the language designer will spot the systematic nature behind these and refactor the language to reflect this deeper understanding of the domain. Language size and complexity is reduced. Over time, the amplitude of these changes in language size and complexity (the error bars in Fig. 4.1) should become smaller, and the language size and complexity should approach a stable level (*ss* in Fig. 4.1).

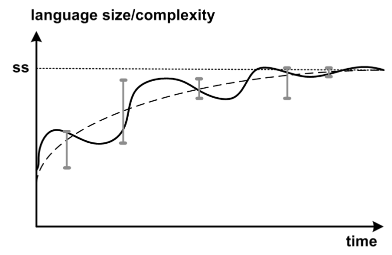


Figure 4.1: Iterating towards a stable language over time. It is a sign of trouble if the language complexity does not approach some kind of stable state over time.

Component Architecture: A nice example of spotting a systematic nature behind a set of special cases was the introduction of data replication as a core abstraction in the architecture DSL (we also discuss this in Section **??**). After modeling a number of message based communication channels, we noticed that the interfaces all had the same set of methods, just for different data structures. At one point we saw the pattern behind it and created new linguistic abstractions: data replication. ◀

4.1.3 *Co-evolve Concepts and Language*

In cases where you perform a real domain analysis, i.e. when you have to find out which concepts the language should contain, make sure you evolve the language in real time as you discuss the concepts.

Defining a language requires formalization. It requires becoming very clear and unambiguous about the concepts that go into the language. In fact, building the language, because of the need for formalization, helps you become clear about the domain abstractions in the first place. Language construction acts as a catalyst for understanding the domain! I recommend actually building a language in real time as you analyze your domain.

Refrigerators: This is what we did in the cooling language. Everybody learned a lot about the possible structure of refrigerators and the limited feature combinations (based on limitations imparted by how some of the hardware devices work). ◀

To make this feasible, your DSL tool needs to be lightweight enough so support language evolution during domain analysis workshops. Turnaround time should be minimal to avoid overhead.

Refrigerators: The cooling DSL is built with Xtext. Xtext allows very fast turnaround regarding grammar evolution, and, to a lesser extent, scopes, validation and type systems. We typically evolved the grammar in real-time, during the language design workshops, together with the domain experts. We then spent a day offline finishing scopes, constraints, the type system as well as the interpreter. ◀

4.1.4 *Let People Do What they are Good At*

DSLs offers a chance to let everybody do what they are good at. There are several clearly defined roles, or tasks, that need to be done. Let met point out two, specifically.

Experts in a specific target technology can dig deep into the details of how to efficiently implement, configure and operate that technology. They can spend a lot of time testing, digging and tuning. Once they found out what works best, they can put their knowledge into platforms and execution engines, efficiently spreading the knowledge across the team. For the latter task, they will collaborate with generator experts and language designers – our second example role.

Component Architecture: In building the language, an OSGi expert was involved in building the generation templates. ◀

The language designer works with domain experts to define abstractions, notations and constraints to accurately capture domain knowledge. The language designer also works with the architect and the platform experts in defining code generators or interpreters. For the role of the language designer, be aware that there needs to be some kind of predisposition in the people who do it: not everybody is good at "thinking meta", some people are more skewed towards concrete work. Make sure you use "meta people" to do the "meta work". And of course, the language designer must able be fluent with the DSL tool used in the project.

There's also a flip side here: you have to make sure you actually do have people on your team who are good at language design, know about the domain and understand target platforms. Otherwise the benefits promised by using DSLs may not materialize.

4.1.5 *Domain Users vs. Domain Experts*

When building business DSLs, people from the domain can play two different roles. They can either participate in the domain analysis and the definition of the DSL itself, or they can use the DSL to create domain-specific models or programs.

It is useful to distinguish these two roles explicitly. The first role (language definition) must be filled by a domain *expert*. These are people who have typically been working in the domain for a long time, often in different roles, and who have a deep understanding of the relevant concepts, which they are able to express precisely, and maybe even formally. The second group of people are the domain *users*. They are of course familiar with the domain, but they are typically not as experienced as the domain experts

This distinction is relevant because you want to work with the domain *experts* when defining the language, but you want to build a language that is suitable to be used by the domain *users*. If the experts are too far ahead of the users, the users might not be able to "follow", and you will not be able to roll out the language to the actual target audience.

Hence, make sure that when defining the language, you actually cross-check with real domain users whether they are able to work with the language.

Pension Plans: The core domain abstractions were contributed by Herman. Herman was the most senior pension expert in the company. In workshops we worked with a number of other domain users who didn't have as much experience. We used them to validate that our DSL would work for the average future user. Of course they also found actual problems with the language, so they contributed to the evolution of the DSL beyond just acting as guinea pigs. ◄

4.1.6 DSL as a Product

The language, constraints, interpreters and generators are usually developed by one (smaller) group of people and used by another (larger) group of people. To make this work, consider the "language stuff" a product developed by one group for use by another. Make sure there's a well defined release schedule, development happens in short, predefined increments, requirements and issues are reported and tracked, errors are fixed reasonably quickly, there is ample documentation and there's support staff available to help with problems and the unavoidable learning curve. These things are critical for acceptance!

A specific best practice is to exchange people: from time to time, make application developers part of the language team to appreciate the challenges of "meta", and make people from the language development team participate in actual application development to make sure they understand if and how their work products suit the people who do the actual application development.

Embedded C: One of our initial proof-of-concept projects didn't really work out very well. So in order to try out our first C extensions and come up with a showcase for an upcoming exhibition, the language developers "had" to build the proof-of-concept themselves. As it turned out, this was really helpful. We didn't just find a big number of bugs, we also experienced first-hand some of the usability challenges of the system at the time. It was easy for us to fix, because it was us who experienced the problems in the first place. ◀

4.1.7 *Documentation is still necessary*

Building the DSLs and execution engines is not enough to make the approach successful. You have to communicate to the users how to use these things in real-world contexts. Specifically, here's what you have to document: the language structure and syntax, how to use the editors and the generators, how and where to write manual code and how to integrate it into generated code, as well as platform/framework decisions (if applicable).

Please keep in mind that there are other media than paper. Screencasts, videos that show flip chart discussions, or even a regular podcast that talks about how the tools change are good choices, too. Also keep in mind that hardly anybody reads reference documentation. If you want to be successful, make sure the majority of your documentation are example-driven or task-based tutorials.

Component Architecture: The documentation for the component architecture DSL contains a set of example applications. Each of them guides a new user through building an increasingly complex application. It explains installation of the DSL into Eclipse, concepts of the target architecture and how they map to language syntax, use of the editor and generator, as well as how to integrated manually written code into the generated base classes. ◀

4.2 *Using DSLs*

4.2.1 *Reviews*

A DSL limits the user's freedom in some respect: they can only express things that are within the limits of DSLs. Specifically, low-level implementation decisions are not under a DSL user's control because they are handled by the execution engine.

However, even with the nicest DSL, users can still make mistakes, the DSL users can still misuse the DSL (the more expressive the DSL, the bigger this risk). So, as as part of your development process, make sure you perform regular model reviews. This is critical especially to the adoption phase when people are still learning the language and the overall approach.

Reviews are easier on DSL level than on code level. Since DSL programs are more concise and support better separation of concerns than their equivalent specification in GPL code, reviews become more efficient.

If you notice recurring mistakes, things that people do in a "wrong" way regularly, you can either add a constraint check that detects the problem automatically, or (maybe even better) consider this as input to your language designers: maybe what the users expect is actually correct, and the language needs to be adapted.

4.2.2 *Compatible Organization*

Done right, using DSLs requires a lot of cross-project work. In many settings the same language (module) will be used in several projects or contexts. While this is of course a big plus, it also requires that the organization is able to organize, staff, schedule and pay for cross-cutting work. A strictly project-focused organization has a very hard time finding resources for these kinds of activities. DSLs, beyond the small ad-hoc utility DSL, are very hard to adopt in such environments.

In particular, make sure that the organizational structure, and the way project cost is handled, is compatible with cross-cutting activities. Any given project will not invest into assets that are reusable in other projects if the cost for developing this asset is billed only to the particular project. Assets that are useful for several projects (or the company as a whole) must also paid for by those several projects (or the company in general).

4.2.3 *Domain Users Programming?*

Technical DSLs are intended for use by programmers. Application domain DSLs are targeted towards domain users, non-programmers who are knowledgabe in the domain covered by the DSL. Can they actually work with DSLs?

In many domains, usually those that have a scientific or mathematical touch, users can precisely describe domain knowledge. In other domains you might want to shoot for a somewhat lesser goal. Instead of expecting domain users and experts to independently specify domain knowledge using a DSL, you might want to pair a developer and a domain expert. The developer can help the domain expert to be precise enough to "feed" the DSL. Because the notation is free of implementation clutter, the domain expert feels much more at home than when staring at GPL source code.

Initially, you might even want to reduce your aspirations to the point where the developer does the DSL coding based on discussions with domain users, but then showing them the resulting model and asking confirming or disproving questions about it. Putting knowledge into formal models helps you point out decisions that need to be made, or language extensions that might be necessary.

Executing the program, by generating code or running some kind simulator can also help domain users understand better what has been expressed with the DSL.

If you are not able to teach a business domain DSL to the domain users, it might not necessarily be the domain users' fault. Maybe your language isn't really suitable to the domain. If you encounter this problem, take it as a warning sign and consider changing the language.

4.2.4 DSL Evolution

A DSL that is successfully used will have to evolved. Just as for any other software artifact, requirements evolve over time and the software has to reflect these changes. In the context of DSLs, the changes can be driven by several different concerns:

Target Platform Changes The target platform may change because of the availability of new technologies that provide better performance, scalability or usability. Ideally, no changes to either the language or the models are necessary: a new execution engine for the changed target platform can be created. In practice it is now always so clean: the DSL may make assumptions about the target platform that are no longer true for the changed or new platform. These may have to be removed from the languages and existing models. Also, the new platform may support different execution options, and the existing models do not contain enough information to make the decision which option to take. In this case, additional annotation models may become necessary[2].

[2] Despite the caveats discussed in this paragraph, the target platform change is typically relatively simple to handle.

Domain Changes As the domain evolves, it is likely that the language has to evolve as well[3]. The problem then is: what do you do with existing models? You have two fundamental changes: keep the old language around and don't change the models or evolve the existing models to work with the new language. The former is often not really practical, especially in the face of several such changes.

[3] If you use a lot of in-language abstraction or a standard library, you may be lucky and the changes can be realized without changes to the language.

The degree to which evolving existing models is painful depends a lot on the nature of the change[4]. The most pragmatic approach keeps the new version of the language backward compatible, so that existing models can still be edited and processed. Under this premise, adding new language concepts is never a problem. However, you must never just delete existing concepts or change them in an incompatible way. Instead, these old concepts should be marked as *deprecated*, and the editor will show an corresponding warning in the IDE. The IDE may also provide a quick fix to change the old, deprecated concept to a new (version of the) concept, if such a mapping is straight forward. Otherwise the migration must be done manually. If you have access to *all* models, you may also run a batch transformation during a quiet period to migrate them all at once. Notice that, although deprecation has a bad rep from programming languages where deprecated concepts are never removed, this is not necessarily comparable to DSLs: if, after a while, people still use the

[4] It also depends a lot on the DSL tool. Different tools support model evolution in different ways.

deprecated concepts, you can have the IDE send an email to the language developers who can then work with the "offending user" to migrate the programs.

Note that for the above approach to work, you have to have a structure process for versioning the languages and tools, otherwise you will end up in version chaos quickly.

DSL Tool Changes The third change is driven by evolution of the DSL tool. Of course, the language definition (and potentially, the existing models) may have to evolve if the DSL tool changes in an incompatible way (which, one could argue, it shouldn't!). This is similar to every other tool, library of framework you may use. People seem particularly afraid of the situation where they have to switch to a completely new DSL tool because the current one is no longer supported, or a new one is just better. Of course it is very likely that you'll have to completely redo the language definitions: there is no portability in terms of language definitions among DSL tools (not even among those that reside on Eclipse). However, if you had designed your languages well you will likely be able to *automatically transform existing models* into the new tool's data structures[5].

[5] It is easy to see that over time the real value is in the *models*, and not so much in the language definitions and IDEs. Rebuilding those is hence not such a big deal, especially if we consider that a new, better tool likely requires less effort to build the languages.

One central pillar of using DSLs is the high degree to which they support separation of concerns and expressing domain knowledge at a level of abstraction that makes the domain semantics obvious, avoiding complex reverse engineering problems. Consequently you can generate all kinds of artifacts from the models. This characteristic also means that it is relatively straight forward to write a generator that creates a representation of the model in a new tool's data structures[6].

[6] For example, we have built a generic MPS to EMF exporter. It works for meta models as well as for the models.

4.2.5 *Avoiding Uncontrolled Growth and Fragmentation*

If you use DSLs successfully, there may be the danger of uncontrolled growth and diversification in languages with the obvious problems for maintenance, training and interoperability[7]. To avoid this, there is an organizational technical approach.

[7] While this may become a problem, this may also become a problem with libraries or frameworks ... the solution is also the same, as we will see.

The organizational approach requires putting in place governance structures for language development. Maybe developers have to coordinate with a central entity before they are "allowed" to define a new language. Or an open-source like model is used, where languages are developed in public and the most successful ones will strive and attract contributions. Maybe you want to limit language development to some central "language team"[8]. Larger organizations in which uncontrolled language growth and fragmentation might become a problem are likely to already have established processes for coordinating reusable or cross-cutting work. You should just plug into these pro-

[8] Please don't overdo this! Don't make it a bureaucratic nightmare to develop a language!

cesses.

The technical approach (which should be used in together with the organizational one) exploits language modularization, extension and composition. If (parts of) languages can be reused, the drive to develop something completely new (that does more or less the same as somebody else's language) is reduced. Of course this requires that language reuse actually works with your tool of choice. It also requires that the potentially reusable languages are robust, stable and documented – otherwise nobody will use them. In an large organization I would assume that a few language will be strategic: aligned with the needs of the whole organization, well-designed, well tested and documented, implemented by a central group, used by many developers and reusable by design[9]. In addition, small teams my decide to develop their own, smaller languages or extensions, reusing the strategic ones. Their focus is much more local, and the development requires much less coordination.

[9] The development of these languages should be governed by the organizational approach discussed above