MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

Part I

DSL Design

1
Core Design Dimensions

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

DSLs are languages with high expressivity for a specific, narrow problem domain. They are a powerful tool 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, completely, and at the right abstraction level, DSL design is also hard. In this chapter we present a framework for describing and characterizing external domain specific languages. We identify eight design dimensions that span the space within which DSLs are designed: expressivity, coverage, semantics, separation of concerns, completeness, large-scale model structure, language modularization and syntax. We illustrate the design alternatives along each of these dimensions with examples from our case studies.

This part of the book presents a conceptual framework for the description of DSL design, based on eight dimensions: expressivity, coverage, semantics and execution, separation of concerns, structuring programs, language modularity, and concrete syntax. These dimensions provide a vocabulary for describing and comparing the design of existing DSLs. While the chapter does not contain a complete methodology for designing new DSLs, the framework does highlight the options designers should consider. We also describe drivers, or forces, that lead to using one design alternative over another one.

1.1 Programs, Languages, Domains

Domain-specific languages live in the realm of *programs*, *languages*, and *domains*. We are primarily interested in *computation*. So, let's first consider the relation between programs and languages. Let's define P to be the set of all programs. A *program* p in P is the Platonic representation of some *effective computation* that runs on a universal computer. That is, we assume that P represents the canonical semantic model of all programs and includes all possible hardware on which programs may run. A *language* L defines a structure and notation for *expressing* or *encoding* programs. Thus, a program p in P may have an expression in L, which we will denote p_L. Note that p_{L_1} and p_{L_2} are representations of a single semantic (platonic) program in the languages L_1 and L_2. There may be multiple ways to express the same program in a language L. A language is a *finitely generated* set of program encodings. That is, there must be a finite description that generates all program expressions in the language. As a result, it may not be possible to define all programs in some language L. We denote as P_L the subset of P that can be expressed in L. A translation T between languages L_1 and L_2 maps programs from their L_1 encoding to their L_2 encoding, i.e. $T(p_{L_1}) = p_{L_2}$.

Pension Plans: The pension language can be used to effectively represent pension calculations, but cannot be used to express general purpose enterprise software ◄

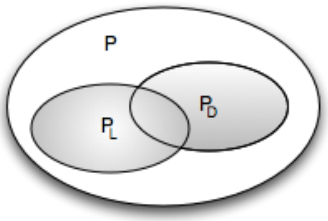

Figure 1.1: Programs, Languages, Domains

NOW, WHAT ARE DOMAINS? There are essentially two approaches to characterize domains. First, domains are often considered as a body of knowledge in the real world, i.e. outside the realm of software. From this *deductive* or *top-down* perspective, a domain D is a body of knowledge for which we want to provide some form of software support. We define P_D the subset of programs in P that implement computations in D, e.g. 'this program implements a fountain algorithm'.

Pension Plans: The pensions domain has been defined this way. The customer had been working in the field of old age pansions 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 some extent, encoded in the source of existing software. ◄

In the *inductive* or *bottom-up* approach we define a domain in terms of existing software. That is, a domain D is identified as a subset P_D of P, i.e. a set of programs with common characteristics or similar purpose. Often, such domains do not exist outside the realm of software.

A special case of the inductive approach is where we define a domain as a subset of programs of a specific P_L instead of the more general set P. In this special case we can often clearly identify the commonality between programs in the domain, in the form of their consistent use of a set of domain-specific patterns or idioms.

Embedded C: The extensions to the C programming language are defined bottom-up. Based on idioms commonly used when using C for a given class of problems, linguistic abstractions have been defined that provide a "shorthand" for those idioms. These linguistic abstractions form the basis of the language extensions. ◄

The above example 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 to expressing refrigerator cooling programs for a very specific organization. No claim is made for widespread usefulness of the DSL. However, it perfectly fits into the way cooling algorithms are described and implemented in that particular organization. ◄

Whether we take the deductive or inductive route, we can ultimately identify a domain D by a set of programs P_D. 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). A *domain-specific language* L_D for D is a language that is *specialized* to encoding P_D programs. That is, L_D is more efficient in some respect in representing P_D programs. Typically, such a language is *smaller* in the sense that P_{L_D} is a strict subset of P_L for a less specialized language L.

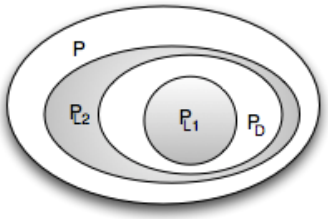

Figure 1.2: Languages L1 and L2 under-approximate and over-approximate domain D.

The crucial difference between languages and domains is that the former are finitely generated, but that the latter are arbitrary sets of programs the membership of which is determined by a human oracle. This difference defines the difficulty of DSL design: finding regularity in a non-regular domain and capturing it in a language. The resulting DSL represents an explanation or interpretation of the domain, and often requires trade-offs by under- or over-approximation (Figure 1.2).

1.1.1 *Programs as Trees of Elements*

Programs are represented in two ways: concrete syntax and abstract syntax. 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

Users use the concrete syntax as they write or change programs. The abstract syntax is a data structure that contains all the data expressed with the concrete syntax, but without the notational details. The abstract syntax is used for analysis and downstream processing of programs.

way round. User editing gestures directly change the abstract syntax, the concrete syntax being a mere projection that looks and feels like text if a textual projection is used. SDF and Xtext are parser-based, MPS is projectional.

While concrete syntax modularization and composition can be a challenge (as discussed in Section 1.8.5), we will illustrate most language design concerns based on the abstract syntax.

The abstract syntax of programs are primarily trees of program *elements*. 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.

A program may be composed from several program *fragments*. A fragment is a standalone tree. E_f is the set of program elements in a fragment f.

A LANGUAGE l defines a set of language concepts C_l and their relationships[1]. In a fragment, each element e is an instance of a concept c defined in some language l.

[1] 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.

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

We define the *concept-of* function *co* to return the concept of which a program element is an instance: $co \Rightarrow element \rightarrow concept$. 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: $fo \Rightarrow element \rightarrow fragment$.

We also define the following sets of relations between program elements. Cdn_f is the set of parent-child relationships in a fragment f. Each $c \in C$ has the properties *parent* and *child*.

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.

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 `LocalVariabedRef` 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 to language concepts. 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 contains a list of statements. ◀

An important concept in LMR&C is the notion of independence. An *independent language* does not depend on other languages. An independent language l can be defined as a language for which the following hold:

$$\forall r \in Refs_l \mid lo(r.to) = lo(r.from) = l \tag{1.1}$$
$$\forall s \in Inh_l \mid lo(s.super) = lo(s.sub) = l \tag{1.2}$$
$$\forall c \in Cdn_l \mid lo(c.parent) = lo(c.child) = l \tag{1.3}$$

An *independent fragment* is one where all non-containing cross-references stay within the fragment (1.4). By definition, an independent fragment has to be expressed with an independent language (1.5).

$$\forall r \in Refs_f \mid fo(r.to) = fo(r.from) = f \tag{1.4}$$
$$\forall e \in E_f \mid lo(co(e)) = l \tag{1.5}$$

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. 1.3). 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. ◀

We also distinguish *homogeneous* and *heterogeneous* fragments. A homogeneous fragment is one where all elements are expressed with the same language:

$$\forall e \in E_f \mid lo(e) = l \tag{1.6}$$
$$\forall c \in Cdn_f \mid lo(c.parent) = lo(c.child) = l \tag{1.7}$$

Embedded C: A program written in plain C is homogeneous. All program elements are instances of the C language. Using the

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 1.3: A `BuildingBlockRef` references a hardware element from within a cooling algorithm fragment.

statemachine language extension allows us to embed state machines in C programs. This makes the respective fragment heterogeneous (see Fig. 1.4). ◀

```
module CounterExample from counterd imports nothing {

  var int theI;

  var boolean theB;

  var boolean hasBeenReset;

  statemachine Counter {
    in start() <no binding>
       step(int[0..10] size) <no binding>
    out someEvent(int[0..100] x, boolean b) <no binding>
        resetted() <no binding>
    vars int[0..10] 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 {  }
      }
  } end statemachine

  var Counter c1;

  exported test case test1 {
    initsm(c1);
    assert(0) isInState<c1, initialState>;
    trigger(c1, start);
    assert(1) isInState<c1, countState>;
  } test1(test case)
}
```

Figure 1.4: 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`) are used inside a `TestCase`.

1.1.2 *Domain Hierarchy*

The subsetting of domains naturally gives rise to a hierarchy of domains (Fig. 1.5). 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 possible 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. At the top of the hierarchy we find singleton domains that consist of a single program (a non-interesting boundary

case). Languages are typically designed for a particular D. Languages for D_0 are called general-purpose languages. Languages for D_n with $n > 0$ become more domain-specific for growing n.

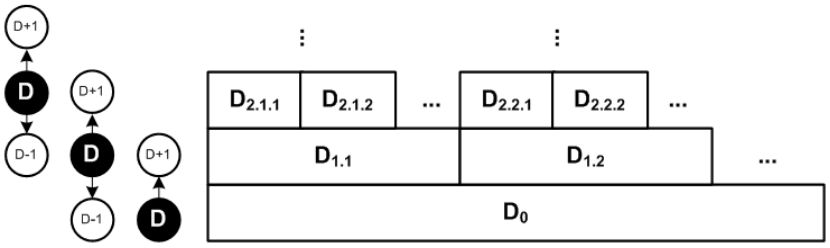

Figure 1.5: Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

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 $D_{2.1.1}$. ◀

1.1.3 *Model Purpose*

We have said earlier that there can be several languages for the same domain. Deciding which concepts should go into a particular language for D, and at which level of abstraction or detail, is not always obvious. The basis for the decision is to consider the *model purpose*. Models, 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 or platform independent specification of functionality[2]. 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.

[2] Communication among humans could be another model purpose. However, we consider languages used only for human-to-human communication outside the scope of this book, because they don't have to be formally defined to achieve their goal.

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. The state machines in the statemachines language have an additional model purpose: model checking, i.e. proofing certain properties about the state machines (e.g. proofing that a certain state is definitely going to be reached after some event occurs). To enable this, actions used in the state machines

are further limited. ◀

Refrigerators: The model purpose is the generation of efficient implementation code for various different target platforms. 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 or 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. ◀

1.1.4 Parsing vs. Projection

There are two main approaches for implementing external DSLs. The traditional approach is parser-based. 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 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 parsers. 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. Projectional editing is well known from graphical editors, virtually all of them use this approach. However, they can also be used for textual syntax. Example tools in this category include the Intentional Domain Workbench (`http://intentsoft.com`) and JetBrains MPS.

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 tradeoffs in the chapter on language implementation). However, we will point out if and when there are different DSL design options depending on which of the two approaches is used.

While in the past projectional text editors have gotten a bad reputation, 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.

TODO: ref

1.1.5 Design Dimensions

There are typically many differnet languages suitable for expressing a particular program. In DSL design we are looking for the *optimal* language for expressing programs in a particular domain. There are multiple dimensions in which we can optimize language designs. Often it is not possible to maximize along all dimensions; we have to find a trade-off between properties. We have identified the following technical dimensions of DSL design: *expressivity*, *coverage*, *semantics and execution*, *separation of concerns*, *completeness*, *structuring programs*, *language modularity*, and *concrete syntax*. In the following sections we will examine each of these dimensions in detail.

1.2 Expressivity

One of the fundamental advantages of domain-specific languages 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 domain of application 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 ca. five times shorter than the C version that is generated from them. ◀

While it is always possible to produce short but incomprehensible programs, overall, shorter programs require less effort to read and write than longer programs, and should therefore be more efficient in software engineering. We will thus assume that, all other things being equal, shorter programs are preferable over longer programs.[3].

The Kolomogorov complexity[4] of an object is the smallest program in some description language that produces the object. In our case the objects of interest are programs in P and we are interested in designing languages that minimize the size of encodings of programs. We use the notation $|p_L|$ to indicate the size of program p as encoded in language L[5]. 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.

A language L_1 is *more expressive* than a language L_2 ($L_1 \prec L_2$),

[3] 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.

[4] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, third edition edition, 2008

[5] 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.

$$\text{if for each } p \in P_{L_1} \cap P_{L_2},\ |p_{L_1}| < |p_{L_2}|.$$

Typically, we need to qualify this statement and restrict it to the domain of interest.

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

Note that optimizing a DSL too far towards conciseness may limit the DSL's ability to cover a substantial part of the relevant programs in the domain, making the DSL useless. We discuss coverage in Section 1.3.

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[6].

[6] We discuss this aspect in the section on completeness (Section 1.6).

DSLs ARE 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. 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. 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.

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 knowledge about the domain. ◀

The same is true for the act of *building* the DSL. The domain has to be scoped, fully explored and systematically structured to be able to build a language that covers that domain. This leads to a deep understanding of the domain itself.

Refrigerators: Building the cooling DSL has helped the thermodynamicists and software developers to understand the domain, its degrees of freedom and the variability in refrigerator hardware and cooling algorithms. 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. ◀

1.2.1 *Expressivity and the Domain Hierarchy*

In the definition of expressivity above we are comparing arbitrary languages. The central idea behind domain-specific languages is that progressive specialization of the domain enables progressively more 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-pointing integer arrays, two typical idioms for state machine implementation in C. Program analysis, such as finding states that can never be left because no transition leads out of them, is trivial. ◀

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 [7]. ◀

[7] E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007

1.2.2 *Linguistic vs. In-Language Abstraction*

Linguistic Abstraction By making the concepts of D first class members of a language L_D, i.e. by defininining 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[8] idioms or patterns are required to express interesting programs in D. Consider the two examples of loops in a Java-like language:

[8] 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 = ...
List<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 (we presume) the order of the elements in the `List` 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 behaviour uses better linguistic abstractions,

because it is clear without analysis that the first loop can be parallelized and the second cannot. The decision can simply be based on the language concept used (`for` vs. `seqfor`)

```
for (int i in arr) {
    sum += i;
}
```

```
seqfor (int i in arr) {
    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.

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. ◀

Embedded C: Another example of how linguistic abstraction is advantageous is static detection of invocations on optional ports as shown in Fig. 1.7. ◀

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 bad because it limits the degrees of freedom available to a transformation engine. In the example above, the loop A 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 pointing 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. ◀

Embedded C: Another good example is optional ports in components. Components (see Fig. 1.6) 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 compartible interfaces. Required ports my be optional, so for a given instance, it 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. How-

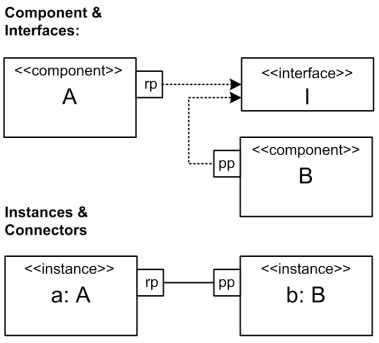

Figure 1.6: Example component diagram. The top half defines components, their ports and the relationship of these ports to interfaces. The bottom half shows instances whose ports are connected by a connector.

ever, an if statement can contain any arbitrary boolean expression as its condition (e.g. `if (isConnected(port) || true) { port.doSomething(); }`). So checking *statically* that the invocation only happens if the port is connected is impossible. An better solution based on linguistic abstraction is to introduce a new language concept that checks for a connected port directly: `with port (port) { port.doSomething(); }`. The `with port` statement doesn't use a complete expression as its argument, but only a reference to a an optional required port (Fig. 1.7). This way, the IDE can check that an invocation on a required optional port `port` is only done inside of a `with port` statement on that same port. ◀

```
exported component AnotherDriver extends Driver {
  ports:
    requires optional ILogger logger
    provides IDriver cmd
  contents:
    field int count = 0

    int setDriverValue(int addr, int value) <- op cmd.setDriverValue {
      with port (logger) {
        logger.log("some error message");
      } with port
      return 0;
    }
}
```

Figure 1.7: The `with port` statement is required to surround an invocation on an optional, required port. If the port is not connected for the current instance, the code inside the `with port` is not executed. It acts as an `if` statement, but since it cannot contain a complete expression, the correct use of the `with port` statement can be statically checked.

In-Lanuguage Abstraction Conciseness can also be achieved by a language providing facilities to define new (non-linguistic) abstractions in programs. 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 tools — at which point they should be refactored into linguistic abstractions.

GPL concepts for building new abstractions include procedures, classes, or functions and higher-order functions.

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, structs, enums) 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 *designer* to provide functionality to language users. Instead of adding language features, a standard library is deployed along with the language to all its users. 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 `rpm` can be specified. The set of properties available for the various building blocks is defined via a standard library and is not part of the language (Fig. 1.8). This way, if a new device driver supports a new property, it can be made available to cooling programs without changing and redeploying the language. ◀

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

Linguistic vs. In-Language Abstraction A language that contains linguistic abstractions for all relevant domain concepts is simple to transform; 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. 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 own set of abstractions in the library. It should be kept in mind that in-language abstraction only works if the transformation of these abstractions is not specific to the abstraction. In such case, linguistic abstraction is better suited.

Modular language extension, as discussed later in Section 1.8.2, provides a middle ground between the two approaches. A language can be flexibly extended, while retaining the advantages of linguistic abstraction.

1.2.3 *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.

Doing 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. 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 deprecatio 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..

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[9]. 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 evolution should be reduced[10].

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

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

Using a set of well-isolated viewpoint-specific DSLs prevents rip-

pling effects on the overall model in case something changes in one DSL.

1.2.4 *Configuration vs. Customization*

Note the difference between configuration and customization. A customization DSL provides a vocabulary which you can creatively combine into sentences of potentially arbitrary complexity. A configuration DSL consists of a well-defined set of parameters for which users can specify values. Configuration languages are more limited, since you cannot easily express instantiation and the relationship between things. However, they are also typically less complex. Hence, the more you can lean towards the configuration side, the easier it usually is to build model processors. It is also simpler from the user's perspective, since the apparent complexity is limited.

Think: feature models. We will elaborate on this in the section of product lines

TODO: ref

1.2.5 *Precision vs. Algorithm*

Be aware of the difference between precision and algorithmic completeness. Many domain experts are able to formally and precisely specify facts about their domain (the "what" of a domain) while they are not able to define (Turing-complete) algorithms to implement the system (the "how"). It is the job of software developers to provide a formal language for domain users to express facts, and then to implement generators and interpreters to map those facts into executable algorithms that truthfully implement the facts they expressed. The DSL expresses the "what", the model processor adds the "how". Consider creating an incomplete language (Section 1.6), and have developers fill in the algorithmic details in GPL code.

Pension Plans: Pension rules are declarative in the sense that a set of mathematical equations and calculation rules are defined. The code generator, written by developers, creates a scalable and sufficiently fast optimization from the models. ◀

1.3 *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 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 in 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 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

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

Since P_L can be larger than P_D, $P_D - (P_D - P_L)$ denotes the P_D programs that can be expressed in P_L. Although this equation does not make sense from a set theory perspective (all sets are typically infinite), it does describe the intuitive notion of the coverage ratio.

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 keep coverage full.

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

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. In this case, the remaining parts of D may have to be expressed with code written in D_{-1} (see also Section 1.6). This requires coordination between DSL users and D_{-1} users, if this not the same group of people.

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. Since this is a very typical case, 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. ◀

1.4 Semantics and Execution

Semantics can be partitioned into static semantics and execution semantics. Static semantics are implemented by the constraints and type system rules (and,if you will, the language structure). Execution semantics denote the observable behaviour of a program p as it is executed. In this section we focus on execution semantics unless stated otherwise.

Using a function OB that defines this observable behaviour 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:

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

Equality of the two observable behaviors can be established with a sufficient number of tests, or with model checking and proof in rare cases. 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 behavioural aspects.

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.

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, or a L_{D-1} 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.

1.4.1 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 the DSL Implementation section of this book . Here is a short overview.

TODO: cite

Constraints are simply boolean expressions that establish 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.

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 to C it is trivial to find states that have no outgoing transitions (dead end, Fig. 1.9). 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_1 is transformed into a model expressed in L_2, which is then in turn transformed into a program expressed in L_3[11]. Make sure that *every* program in L_1 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_1 will not be able to act on these; they may have never seen the programs generated in L_2. Again, automated testing is the way to address this issue.

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 evaluation expression is non-trivial, since it can be an arbitrarily complex expression. A type system defines the rules to establish the types of arbitrary expressions, as well as type checking constraints. We cover the implementation of type systems in the DSL implementation part of the book .

TODO: ref

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

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 constraint that checks that there is at least one element. The reason for using this approach is that if the grammar mechanism is used, a possible error message comes from the parser. That error message may read something like *expecting SUCH_AND_SUCH, found SOMETHING_ELSE*. This is not very useful. If a more tolerant (0..*n*) grammar is used, the constraint error message can be made to express a real domain constraint (e.g. *at least one SUCH_AND_SUCH is required, because ...*).

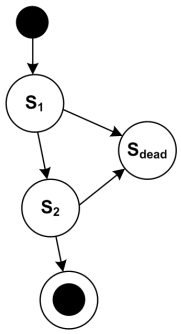

Figure 1.9: An example state machine with a *dead end* state, a state that cannot be left once entered (no outgoing transitions).

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

above, where a type has to be specified explicitly. A type specification expresses the intent that this variable be of type int. Alternatively, a type system could be built to automatically derive the type of the variable declaration, 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 validator 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, validatio of the sub plan will report an error. An alternative design would simply derive 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 based on 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 can then check if the program conforms to this intention. The alternative approach would derive 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. It also anchors 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 the program. 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.

1.4.2 *Transformation*

Transformations define the semantics of a DSL by mapping it to another language. In the context of the domain hierarchy, 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 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-case statements. This is repeated for higher D languages. The semantics of the robot control DSL (Fig. 1.10) is defined by its mapping to state machines and tasks (Fig. 1.11). 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 provided, and a mapping to a distribution middleware is implemented. The formal definition of the semantics are implied by the mapping to the executable code. ◀

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 bottom-up, this works well. For application domain DSLs, and for domains defined 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 as well as test cases or simulators. These can be written in (another part of the) DSL itself; this way, domain users can play with the DSL and write down their expectations in the testing aspect.

Refrigerators: This DSL has a separate viewpoint for defining test cases where domain experts can codify their expectations regarding the behaviour 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. 1.12). The calculations are functional, and the calculation tree can be

```
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 1.10: 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.

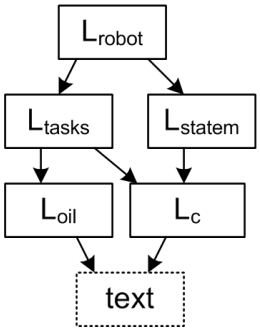

Figure 1.11: Robot control programs are mapped to state machines and tasks. State machines are mapped to C, and tasks are mapped to C as well as to operating system configuration files (a so-called OIL file). In the end, everything ends up in text for downstream processing by existing tools.

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 1.12: 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.

DSL programs may also be mapped to a *different* language, on that is not "under" the DSL in the domain hierarchy, but rather "somewhere on the side". This is often done to establish certain properties about the DSL program. In many cases, these languages are *not* the ones that are used to execute the DSL program, they are specialized formalisms to support verification or proof. In this case, one has to make sure that the representation in both languages is actually the same. We discuss this problem in Section 1.4.6.

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". ◄

TODO: cite

Multi-staged Transformation In cases where the semantic gap between the DSL and the target language is large, it makes sense to introduce intermediate languages so the transformation can be modularized. The overall transformation becomes a chain of subsequent transformations, and appraoch also known as cascading. 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 (compilers have been doing this for a long time). This aspect makes this approach much more useful than

what the pure reuse of languages and transformations suggests. Splitting a transformation into a chain of smaller ones also makes each of them easier to understand and maintain.

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.

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. ◄

As we can learn from compilers, 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.

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. ◄

A special case of a multi-staged transformation is as a preprocessor to a code generator. Here, a transformation is used to reduce the set of used language concepts in a fragment to a minimal core, and only the minimal core is supported in the code generator.

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. 1.13). Once done, the existing generator is run unchanged. You have effectively modularized the emergency stop concern into a separate transformation. ◄

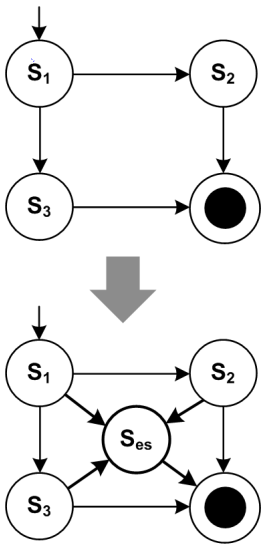

Figure 1.13: A transformation adds an emergency stop feature to a state machine. A new state is added (S_{es}), and a transition from each other state to that new state is added as well. The transition is triggered by the *emergency stop* event (not shown).

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. ◄

Efficiency and Optimization As mentioned, transforming from D to D_{-1} allows the use of sophisticated optimizations, potentially resulting in very efficient code. Because the DSL uses domain-specific abstractions and hence includes a lot of domain semantics, 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 produce more efficient D_{-1} code. However, this requires a competent developer and, usually, a lot of effort *in each case*. A tool 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.

This argument pro tools is also used for garbage collection, transactional memory and optimizing compilers for higher level programming languages.

Care about generated code Ideally, generated code is a throw-away artifact, a bit like object files in a C compiler. However, that's not quite true. When integrating generated code with manually written code, you will have to read the generated code and understand it to some extent. At least during development and test of the generator you may have to debug it. Hence, generated code should use meaningful abstractions, use good names for identifiers, be documented well, and be indented correctly. Making generated code adhere to the same standards as manually written code also helps to diffuse some of the skepticism against code generation that is still widespread in some organizations.

The only exception to this 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.

Note that in complete languages with full coverage (i.e. where 100% of the 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.

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 in-

terface 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 their required ports connected to different component instances (implemeting the same interface) qthere 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. An indirection through function pointers in used. 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. ◀

```
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 1.14: 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 behaviour 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

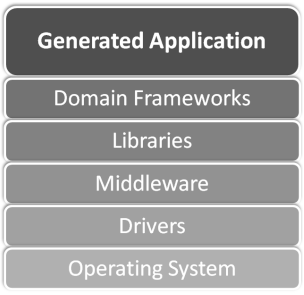

Figure 1.15: Typical layering structure of an application created using DSLs.

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 does open up a new set of options and it is often a very good approach (after all, it's basically the same as compilation, and that's a proven and important technique).

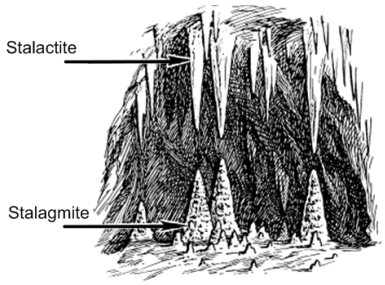

Figure 1.16: Stalagmites and stalactites in limestone caves as a metaphor for a generator and a platform.

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/case statements. If we would generate Java code in an enteprise 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. ◄

1.4.3 *Interpretation*

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.

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 1.7.2). 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 the section on DSL implementation .

TODO: ref

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 "ex-

pands" the calculation tree, and users can inspect all intermediate results. ◀

1.4.4 *Transformation vs. Interpretation*

A primary concern in semantics is the decision 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 appraoch 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.

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[12]. Generally, performance is better in generated environments, since interpreters always imply an additional layer of indirection.

[12] For example, it is not necessary to optimize away calls to empty functions or if statements that always evaluate to `true`

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.

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 editor can be integrated into the executing system.[13].

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

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 GPL VMs such as the JVM, we have not seen this approach used for DSLs.

1.4.5 *Sufficiency*

A fragment is *sufficient for transformation T* if the fragment itself contains all the data for the transformation to be executed. It is *insufficient* if it is not. 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. ◄

1.4.6 *Synchronizing Multiple Mappings*

The approach suggested so far works well if we have only one mapping of a DSL for execution. The semantics implied by the mapping to L_{D-1} can be *defined* to be correct. However, as soon as we transform the program to several different targets in D_{-1} using several different transformations, we have to ensure that the semantics of all resulting programs are identical. In this case we recommend providing a set of test cases that are executed in both the interpreted and generated versions, expecting them to succeed in both. 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.

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.

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. ◄

1.4.7 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 to guide the way the transformation is performed. An annotation model is a separate viewpoint (Section 1.5) 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 contain additional data to guide the decision. Since the data controlling the transformation is embedded in with the core progeam, this is only useful if the DSL user can actually decide which alternative to choose, and if only one alternative should be chose for each program.
- Heuristics, based on patterns, idioms and statistics extracted from the L_D program, can be used to determine the applicable transformation. Codifying these rules and heuristics can be hard though.

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 are expected to be different, since they are the reason for the different transformations in the first place).

1.4.8 Reduced Expressiveness

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 model checking algorithms are available (e.g. using the SPIN or NuSMV model checkers). The same is true for first-order logic, where satisfiability (SAT) solvers[14] can be used to check programs for consistency. If these kinds of analysis are useful for the model purpose, then limiting the expressiveness to the respective formalism may be a good idea, even if it makes expressing

TODO: cite

[14] D. G. Mitchell. A sat solver primer. *eatcs*, 85:112–132, 2005

certain programs in D more cumbersome. Possibly a DSL should be partitioned into several sub-DSLs, where some of them are verifiable and some are not.

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 behaviour 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 tagged 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. This may be a lot of work, or even impossible.

TODO: talk about mapping DSLs down to state machines and reinterpreting the result in the context of the DSL. Should be a result from the LWES project.

1.5 *Separation of Concerns*

A domain D can be composed from different concerns. Fig. 1.17 shows $D_{1.1}$ composed from the concerns A, B and C. To describe a complete program for D, the program needs to address all the concerns.

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

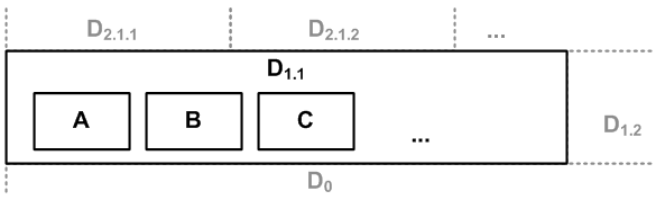

Figure 1.17: A domain may consist of several concerns. A domain is covered either by a DSL that addresses all of these concerns, or by a set of related, concern-specific DSLs.

Figure 1.18: Part A shows an integrated DSL, where the various concerns (represented by different line styles) are covered by a single integrated language (and consequently, one model). Part B shows several viewpoint languages (and model fragments), each covering a single concern. Arrows in Part B highlight dependencies between the viewpoints.

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. A complete program then consists of a set of dependent, concern-specific fragments that relate to each other in a well-defined way. Viewpoints support this separation of domain concerns into separate DSL. Fig. 1.18 illustrates the two different approaches.

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 in a single integrated model. ◀

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[15]. 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. ◀

[15] Z. Hemel, D. M. Groenewegen, L. C. L. Kats, and E. Visser. Static consistency checking of web applications with WebDSL. *JSC*, 46(2):150–182, 2011

1.5.1 *Viewpoints for Concern Separation*

If viewpoints are used, the concern-specific languages, 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.

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

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").

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 fragmwnt. 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.

Viewpoints are also a good fit if the independent fragment is sufficient for a transformation in the domain, i.e. it can be processed without the presence of the additional concerns expressed in sepate viewpoints.

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/file. 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. ◀

1.5.2 *Viewpoints as Annotation Models*

A special case of viewpoint separation is annotation models (already mentioned in Section 1.4.7). An annotation provides additional, often technical or transformation-controlling data for elements in a core program. This is especially useful in a multi-stage transformation (Section 1.4.2) where additional data may have to be specified for the result 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. 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

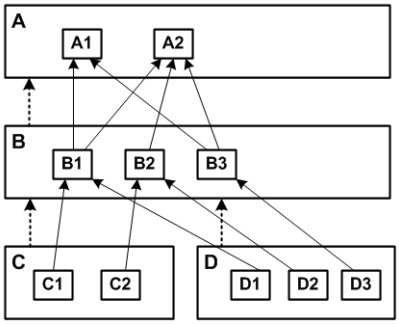

Figure 1.19: Progressive refinement: the boxes represent models expressed with corresponding languages. The dotted arrows express dependencies, whereas the solid arrows represent references between model elements.

name in the original model. ◄

1.5.3 *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 complex systems. Development starts with requirements, proceeds to high-level component design and specification of non-functional properties, and finishes with the implementation of the components. Each of these refinement steps may be expressed with a suitable DSL, realizing the various "refinement viewpoints" of the system (Fig. 1.19). The references between model elements are called traces[16]. 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 (enabled via techniques described in[17]).

[16] W. Jirapanthong and A. Zisman. Supporting product line development through traceability. In *apsec*, pages 506–514, 2005

[17] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations. In *ICMT*, pages 61–76, 2010; and P. Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *SoSyM*, 9(1):7–20, 2010

1.5.4 *Viewpoint Synchronization*

In some cases the models for the various concerns need to be synchronized. This means that when a change happens in a model representing one viewpoint, the models representing other viewpoints must change in a consistent way. It depends on the tools used whether synchronization is feasible: in projectional tools it is relatively easy to achieve, for parser-based systems it can be problematic.

TODO: Say something about on demand, simple listeners, Krzysztofs work in that space and QVTO and look at the example below. Use the ports example instead?

Embedded C: Components implement interfaces. Each component provides an implementation for each method define 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. ◄

1.5.5 *Views on Programs*

In projectional editors it is also possible to store the data for all viewpoints in the same model tree, while showing 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.

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. 1.20). The dependencies can still be edited in this view, but the actual content of the pension plans is not shown. ◄

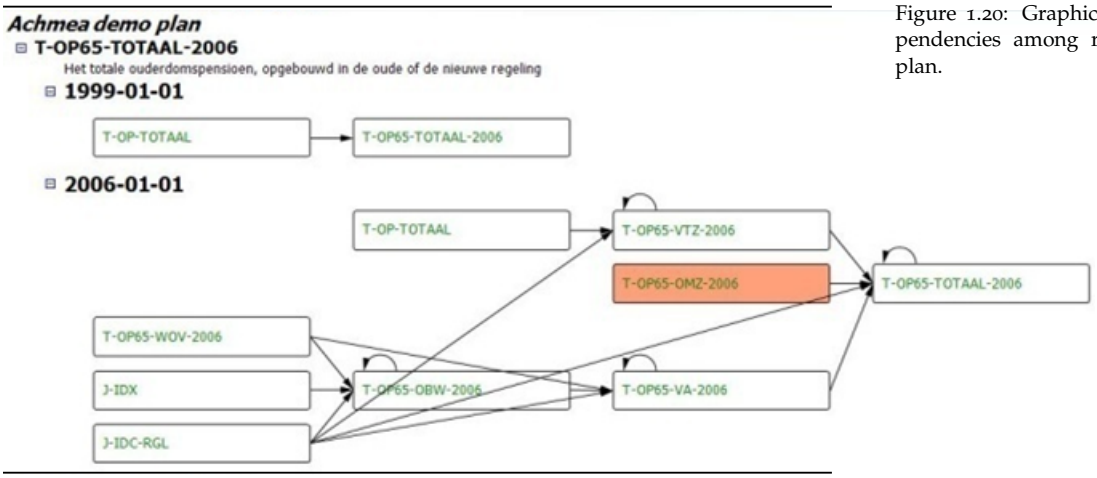

Figure 1.20: Graphical notation for dependencies among rules in a pension plan.

Embedded C: Annotations are used for storing requirements traces and documentation information in the models (Fig. 1.21). The program can be shown and edited with and without requirements traces and documentation text. ◄

```
initialize {
  trace OptionalOutput
  { debugString(0, "state:", "initializing"); }
  ecrobot_set_light_sensor_active(SENSOR_PORT_T::NXT_PORT_S1);
  trace Calibration
  ecrobot_init_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
  trace OptionalOutput
  { debugString(0, "state:", "running"); }
  event linefollower:initialized
}
```

Figure 1.21: 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.