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 ◄

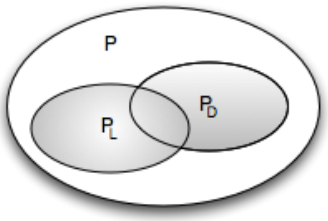

Figure 1.1: Programs, Languages, Domains

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.

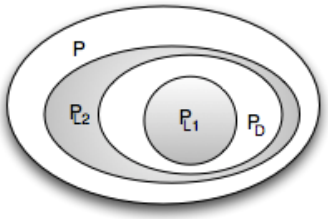

Figure 1.2: Languages L1 and L2 under-approximate and over-approximate domain D.

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. ◀

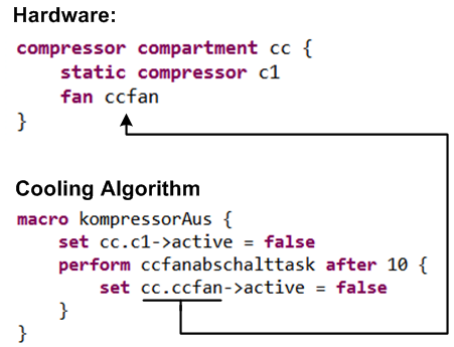

Figure 1.3: A `BuildingBlockRef` references a hardware element from within a cooling algorithm fragment.

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

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.

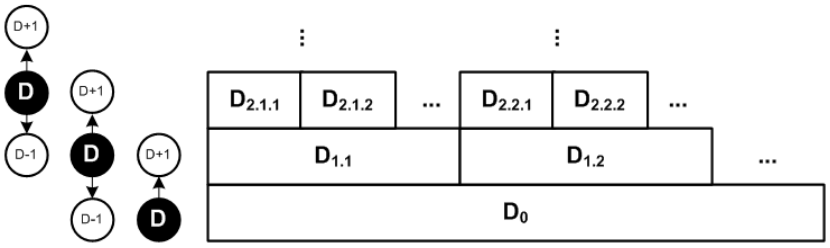

Figure 1.5: Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

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]

[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]

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. ◀

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, chechking whether the port is connected. However, 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 invoca-

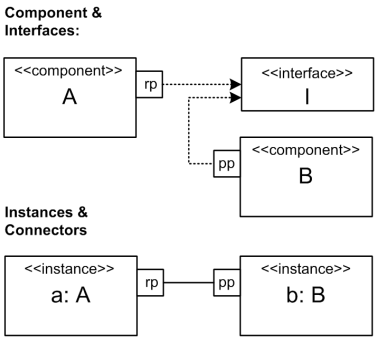

Figure 1.6: Example component diagram. The top half defines components, their ports and the relationship of these ports to interfaces. The bottom half shows instances whose ports are connected by a connector.

tion 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. ◀

TODO: refer tro the above optional port picture

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 rippling 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).

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, $P_D \subseteq P_L$.

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.

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 ...*).

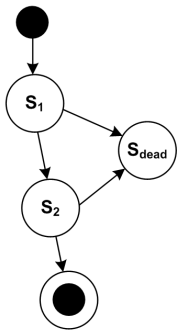

Figure 1.9: An example state machine with a *dead end* state, a state that cannot be left once entered (no outgoing transitions).

[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.

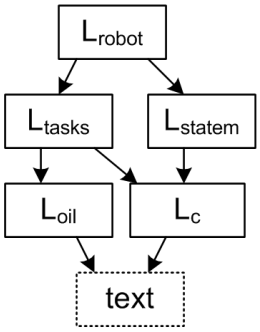

Figure 1.11: Robot control programs are mapped to state machines and tasks. State machines are mapped to C, and tasks are mapped to C as well as to operating system configuration files (a so-called OIL file). In the end, everything ends up in text for downstream processing by existing tools.

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. ◄

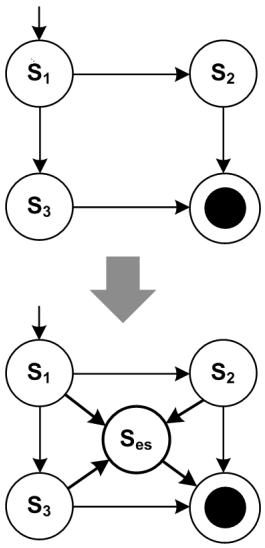

Figure 1.13: A transformation adds an emergency stop feature to a state machine. A new state is added (S_{es}), and a transition from each other state to that new state is added as well. The transition is triggered by the *emergency stop* event (not shown).

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. ◄

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 interface can be provided by different components, implementing the interface differently. Upon translation of the component extension, regular C functions are generated. An outgoing call on a required port has to be routed to the function that has been generated to implement the called interface operation in the target component. Since each component can be instantiated multiple times, and each instance can have 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. ◄

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 imple-

```
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.

mented, rich domain specific platform. It typically consists of middleware, frameworks, drivers, libraries and utilities that are taken advantage of by the generated code.

Where the generated code and the platform meet depends on the complexity of the generator, requirements regarding code size and performance, the expressiveness of the target language and the potential availability of libraries and frameworks that can be used for the task.

In the extreme case, the generator just generates code to populate/configure the frameworks (which might already exist, or which you have to grow together with the generator) or provides statically typed facades around otherwise dynamic data structures. Don't go too far towards this end, however: in cases where you need to consider resource or timing constraints, or when the target platform is predetermined and perhaps limited, code generation 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).

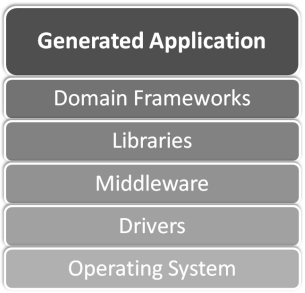

Figure 1.15: Typical layering structure of an application created using DSLs.

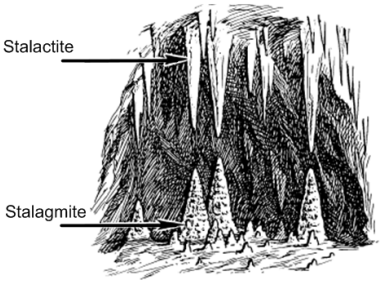

Figure 1.16: Stalagmites and stalactites in limestone caves as a metaphor for a generator and a platform.

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 "expands" 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.

TODO: This is partial evaluation, right?

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 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.

TODO: cite

[14]

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 veri-

fication, 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 appraoch 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).

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

[15]

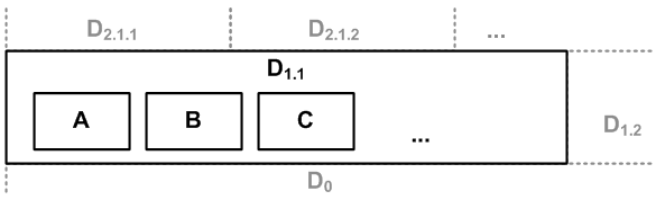

Figure 1.17: A domain may consist of several concerns. A domain is covered either by a DSL that addresses all of these concerns, or by a set of related, concern-specific DSLs.

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.

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. ◀

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.

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").

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

[17] ; and

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

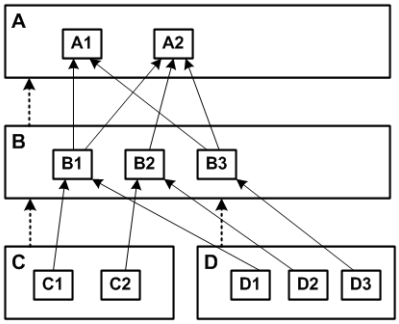

Figure 1.19: Progressive refinement: the boxes represent models expressed with corresponding languages. The dotted arrows express dependencies, whereas the solid arrows represent references between model elements.

synchronization is feasible: in projectional tools it is relatively easy to achieve, for parser-based systems it can be problematic.

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. ◄

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?

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. ◄

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. ◄

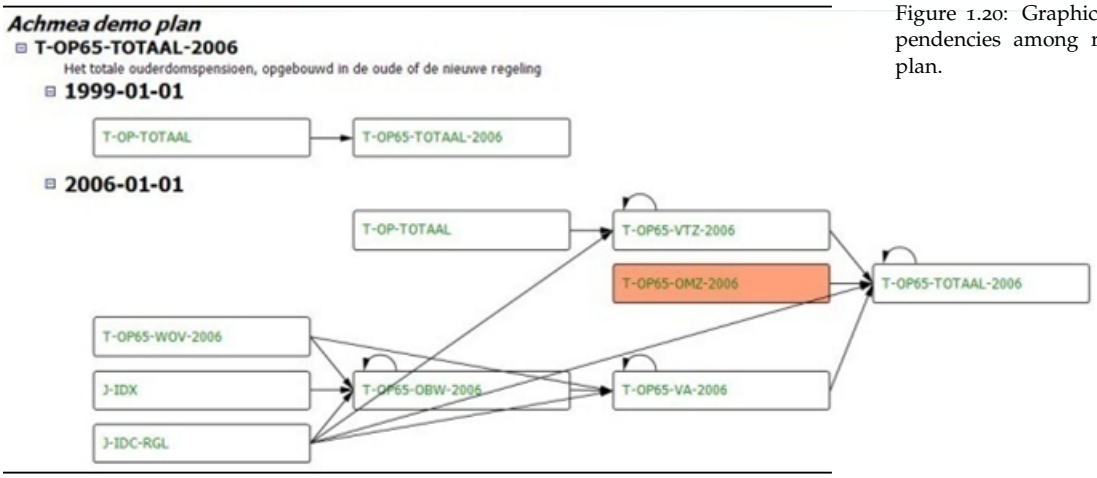

Figure 1.20: Graphical notation for dependencies among rules in a pension plan.

```
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.

1.6 *Completeness*

Completeness refers to the degree to which a language L is able to express *complete* programs. Let us introduce a function G ("code generator") that transforms a program p in L_D to a program q in L_{D-1}. For a complete language, p and q have the same semantics, i.e. $OB(p) == OB(G(p)) == OB(q)$. For incomplete languages where $OB(G(p)) \subset OB(p)$ we have to write additional code in L_{D-1}, to obtain a program in D_{-1} that has the same semantics as intended by the original program in L_D. In cases where we use several viewpoints to represent various concerns of D, the set of fragments written for these concerns must be enough for complete D_{-1} generation. General purpose languages at D_0 are by definition complete.

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

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

Refrigerators: This DSL uses several viewpoints: one to define the structure of a refrigerator, and one to describe the actual algorithm. When generating the C code for the algorithm, both viewpoints are needed, because the structure determines how exactly some of the algorithms are implemented in the generated C code. ◀

1.6.1 *Compensating for Incompleteness*

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

- by calling "black box" code written in L_{D-1}. This requires concepts in L_D for calling D_{-1} foreign functions. No syntactic embedding of D_{-1} code is required.
- by directly embedding L_{D-1} code in the L_D program. This is useful if L_D is an extension of L_{D-1}, or if the tool provide adequate support for embedding the D_{-1} language into L_D programs.

Just "pasting text into a textfield", an approach used by several graphical modeling tools, is not productive, since no syntactic and semantic integration between the languages is provided.

- by using composition mechanisms of L_{D-1} to "plug in" the manually written code into the generated code without actually modifying the generated files (also known as the Generation Gap pattern [18]). Example techniques for realizing this approach include generating a base class with abstract methods (requiring the user to implement them in a manually written subclass) or with empty callback methods which the user can use to customize in a subclass (for example, in user interfaces, you can return a position object for a widget, the default method returns null, default to the generic layout algorithm). You can delegate, implement interfaces, use #include, use reflection tricks, AOP or take a look at the well-known design patterns for inspiration. Some languages provide partial classes, where a class definition can be split over a generated file and a manually written file.

[18]

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

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

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

Component Architecture: This DSL is not complete. Only class skeleton and infrastructure integration code is generated from the

models. The component implementation has to be implemented manually in Java using the Generation Gap pattern. The DSL is used by developers, so writing code in a subclass of a generated class is not a problem. ◀

For DSLs used by domain experts, the situation is different. Usually, they are not able to write D_{-1} code, so other people (developers) have to fill in the remaining concerns. Alternatively, developers can develop a predefined set of foreign functions that can be called from within the DSL. In effect, developers provide a standard library (cf. Section 1.2.2) which can be invoked as black boxes from DSL programs.

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

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

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

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

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

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

To address this issue, make sure there is there a way to make those conventions and idioms interactive. One way to do this is to generate

checks/constraints *against the code base* and have them evaluated by the IDE, for example using Findbugs or similar code checking tools. qIf one fails, an error message is reported to the developer. That error message can be worded by the developer of the DSL, helping the devloper understand what exactly has to be done to solve the problem in the code.

TODO: cite

Broken Promises As part of the definition of a DSL you will implement constraints that validate the DSL program in order to ensure some property of the resulting system (see Section 1.4.1).q For example, you might check dependencies between components in an architecture model to ensure components can be exchanged in the actual system. Of course such a validation is only useful if the manually written code does not introduce dependencies that are not present in the model. In that case the "green light" from the constraint check does not help much.

To ensure that promises made by the models are kept by the (manually written) code, use on of the following two approaches. First, generate code that does not allow violation of model promises. For example, don't expose a factory that allows components to look up and use any other component (creating dependencies), but rather use dependency injection to supply objects for the valid dependencies expressed in the model. Second, use architecture analysis tools (dependency checkers) to validate manually written code. You can easily generate the checking rules for those architecture analysis tools from the models.

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

1.6.2 *Roundtrip Transformation*

Roundtrip transformation means that an L_D program can be recovered from a program in L_{D-1} (written from scratch, or changed manually after generation from a previous iteration of the L_D program). This

is challenging, because it requires reconstituting the semantics of the L_D program from idioms or patterns used in the L_{D-1} code. This is the general reverse engineering problem and is not generally possible, although progress has been made over recent years (see for example [19]).

[19] ; ; and

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

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

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

We generally recommend to avoid (the attempt of building support for) roundtripping.

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

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

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

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

1.7 Fundamental Paradigms

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

1.7.1 Structure

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

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

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

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

Embedded C: As a fundamental extension to C, this DSL contains modules with visibility specifications and imports (Fig. 1.22). Functions, state machines, tasks and all other top-level concepts reside in modules. Header files (which are effectively a poor way of managing symbol visibility) are only used in the generated low level code. ◀

```
module Module1 from HPL.main imports Module2 {

  exported var int aReallyGlobalVar;

  struct aLocallyVisibleStruct {
    int x;
    int y;
  };

  exported int anExportedFunction() {
    return anImportedFunction/Module2();
  } anExportedFunction (function)
}
```

Figure 1.22: Modules are the top-level contruct in this C implementation. Module contents can be exported, which means they are visible to importing modules.

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

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

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

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

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

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

Other concerns influence the design of a partitioning strategy:

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

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

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

Tool Chain Integration integration with existing, file based tool chains. Files may be the unit of checkin/checkout, versioning, branching or permission checking.

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

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

The partitioning scheme may also influence users' team collaboration when editing models. There are two major collaboration models: real-time and commit-based. In real-time collaboration, a user sees his model change when another user changes that same model. Change propagation is immediate. A database-backed repository is often a good choice regarding storage, since the granularity tracked by the repository is the model element. In this case, the partitioning may not be visible to the end user, since they just work "on the repository". This approach is often (at least initially) preferred by non-programmer DSL users.

The other collaboration mode is commit-based where a user's changes only make it to the repository if he performs a *commit*, and incoming changes are only visible after a user has performed an *update*. While this approach can be used with database-backed repositories, it is most often used with file-based storage. In this case, the partitioning scheme is visible to DSL users, because it is those files they commit or update.

This approach tends to be preferred by developers, maybe because well-known versioning tools have used the approach for a long time.

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

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

Embedded C: This DSL adds interfaces and components to C. Components provide or use one or more interfaces. Different components can be plugged in behind the same interface. In contrast to C++, no runtime polymorphism is supported, the translation to plain C maps method invocation to flat function calls. ◀

Refrigerators: Cooling programs can refer to entities defined as part of the refrigerator hardware as a means of accessing hardware elements (compressors, fans, valves). To enable cooling programs to run with different, but similar hardware configurations, the hardware structure can use "trait inheritance", where a hardware trait defines a set of hardware elements, acting as a kind of interface. Other hardware configurations can inherit these traits. As long as cooling programs are only written against traits, they work with any refrigerator that implements the particular set of traits against which the program is written. ◀

Specialization Specialization enables one entity to be a more specific variant of another one. Typically, the more specific one can be used in all contexts where the more general one is expected (the Liskov substitution principle[20]). The more general one may be incomplete, requiring the specialized ones to "fill in the holes". Specialization in the context of DSLs can be used for implementing variants or of evolving a program over time.

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

Pension Plans: The customer using this DSL had the challenge of creating a huge set of pension plans, implementing changes in relevant law over time, or implementing related plans for different customer groups. Copying complete plans and then making adaptations was not feasible for obvious reasons. Hence the DSL provides a way for pension plans to inherit from one another. Calculation rules can be marked *abstract* (requiring overwriting in sub-plans), *final* rules are not overwritable. Visibility modifiers

control which rules are considered "implementation details". ◀

Refrigerators: A similar approach is used in the refrigerator DSL. Cooling programs can specialize other cooling programs. Since the programs are fundamentally state-based, we had to define what exactly it means to override a pumping program. ◀

Types and Instances Types and instances refers to the ability to define a structure that can be parametrized when it is instantiated.

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

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

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

This is especially important in the context of product line engineering and is discussed in and in section of this book.
TODO: ref

Component Architecture: This DSL provides a way of advising component definitions, for example to introduce additional ports from an aspect (Fig. 1.23). An aspect may introduce a port `provided port mon: IMonitoring` that allows a central monitoring component to query the adviced components via the `IMonitoring` interface. ◀

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

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

The other alternative is to make versioning and tracking over time a part of the language. For example, model elements can be tagged

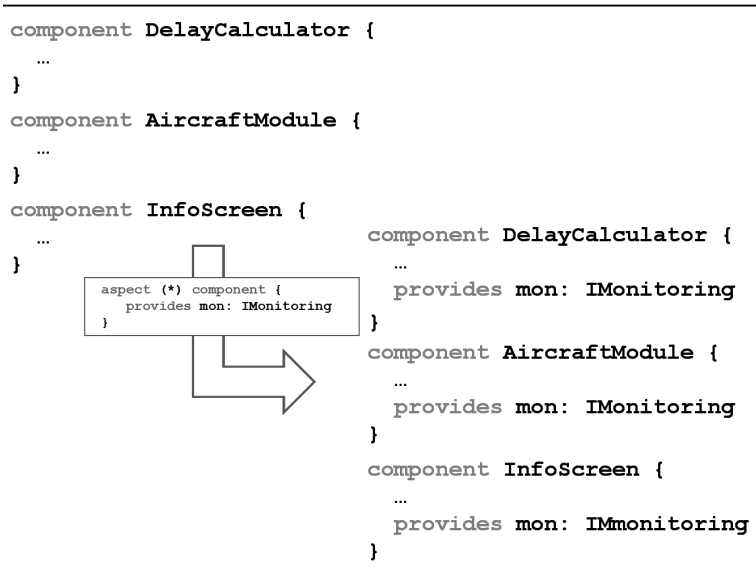

Figure 1.23: The aspect component contributes an additional required port to each of the other components defined in the system.

with version numbers or specify a revision chain by pointing to a previous revision, and enforcing compatibility constraints between those revisions. Instead of declaring explicit versions, business data is often time-dependent, where different revisions of a business rule apply to different periods of time. Support for these approaches can be built directly into the DSL, with various levels of tool support.

Embedded C: No versioning is defined into the DSL. Users work with MPS' integration with popular version control systems. ◀

Component Architecture: Interfaces can specify a `new version of` reference to another interface. If they do so, a constraint enforces that the new version provides at least the same operations as the old version, plus optional additional ones. `new version of` can also be used between components. In this case, the new version has to have the same (or additional) provided ports with the same interfaces or new versions of these interfaces. It must have the same or fewer required ports. Effectively, this means that the new version of something must be replacement-compabible with the old version (the Liskov substitution principle again). ◀

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

1.7.2 *Behavior*

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

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

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

Embedded C: The component extension to provides a similar notion of interfaces, ports and components as in the previous example. However, since here they are directly integrated with C, C expression can be used for pre- and post-conditions of interface operations (Fig. 1.24). ◀

```
c/s interface IDriver {
  int setDriverValue(int addr, int value)
    pre value > 0
}
```

Figure 1.24: Precondiditions can be added to interface operations. They are executed whenever any runnable that implements the particular operation is executed.

This section described some of the most well-known behavioural paradigms that can serve as useful starting points for behaviour descriptions in DSLs.

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

Imperative Imperative programs consist of a sequence of statements, or instructions, that change the state of a program. This state may be local to some kind of module (e.g. a procedure or an object), global (as in global variables) or external (when talking to periphery). Procedural and object-oriented programming are both imperative, using different means for structuring and (in case of OO) specialization. Because of

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

aliasing and side effects imperative programs are expensive to analyse. Debugging imperative programs is straight forward and involves stepping through the instructions and watching the state change.

Embedded C: Since C is used as a base language, this language is fundamentally imperative. Some of the DSLs on top of it use other paradigms. ◀

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

Functional Functional programming uses functions as the core abstraction. A function's return value only depends on the values of its arguments. Functions cannot access global state, no side effects are allowed. Calling the same function several times with the same argument values has to return the same value (that value may even be cached!). No aliasing (through mutable memory cells) is supported, because values are immutable once they are created. Since all dependencies of a computed value are local to a function (the arguments), various kinds of analyses are possible. If assignment to variables is supported, then it uses a form where a variable can only be assigned once.

To create real-world programs, a purely functional language is usually not sufficient, because it cannot affect the environment (after all, this is a side effect). Since there is no state to watch change as the program steps through instructions, debugging can be done by simply showing all intermediate results of all function calls, basically "inspecting" the state of the calculation. This makes building debuggers much simpler. Functional programming is often used for certain parts ("calculation core") of a more complex system.

Pension Plans: The calculation core of pension rules is functional. Consequently, a debugger has been implemented that, for a given set of input data, shows the rules as a tree where all intermediate results of each function call are shown (Fig. 1.25). No "step through" debugger is necessary. ◀

A relevant subset of functional programming is pure expressions (as in `3*2+7 > i`. Instead of calling functions, operators are used. However, operators are just inline-notations for function calls. Usually the operators are hard wired into the language and it is not possible for users to define their own functional abstractions. This is the main differentiator to functional programming in general.

Embedded C: We use expressions in the guard conditions of the state machine extension as well as in pre- and post-conditions

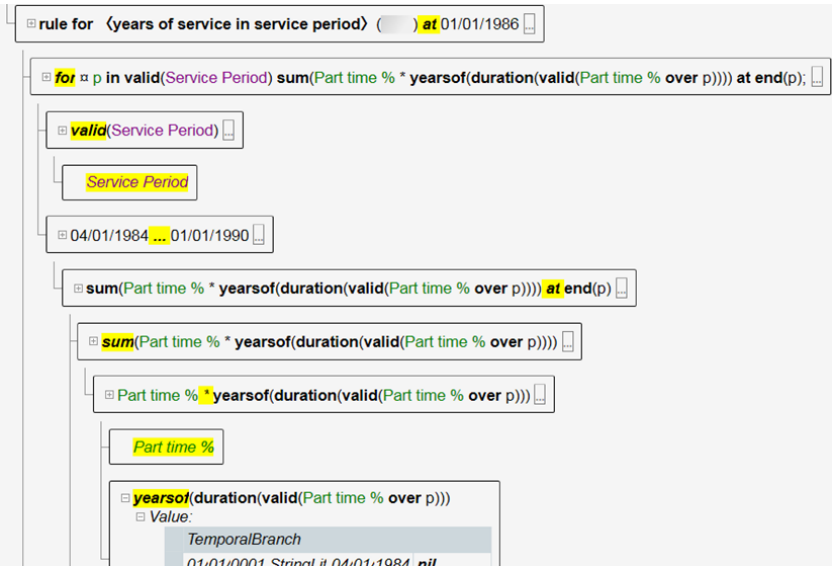

Figure 1.25: Debugging functional programs can be done by showing the state of the calculation, for example as a tree.

for interface operations. In both cases it is not possible to define or call external functions. Of course, C's expression language is reused here. ◀

Declarative Declarative programming can be considered the opposite of imperative programming (and, to some extent, functional programming). A declarative program does not specify any control flow. It does not specify a sequence of steps of a calculation. Instead, a declarative program only specifies what the program should accomplish, not how. This is often done by specifying a set of properties, equations, relationships or constraints. Some kind of evaluation engine then tries to find solutions. The particular advantage of this approach is that it is not predefined how a solution is found, the evaluation engine has a lot of freedom in doing so, possibly using different solutions in different environments, or evolving the approach over time[21]. This large degree of freedom often makes finding the solution expensive — trial and error, backtracking or exhaustive search may be used. Debugging declarative programs can be hard since the solution algorithm may be very complex and possibly not even be know to the user of the language.

Declarative programming has many important subgroups and use cases. For *concurrent programs*, a declarative approach allows the efficient execution of the same program on different parallel hardware. The compiler or runtime system can allocate the program to available computational resources. In *constraint programming*, the programmer specifies constraints between a set of variables. The engine tries to find

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

values for these variables that satisfy all constraints. Solving mathematical equation systems is an example, as is solving sets of boolean logic formulas.

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

Embedded C: This DSL supports presence conditions for product line engineering. A presence condition is a boolean expression over a set of configuration features that determines whether the associated piece of code is present for a given combination of feature selections (Fig. 1.26). To verify the structural integrity of programs in the face of varying feature combination, constraint programming is used (to ensure that there is no configuration of the program where a reference to a symbol is included, but the referenced symbol is not). From the program, the presence conditions and the feature model a set of boolean equations is generated. A solver then makes sure they are consistent by trying to find an example solution that violates the boolean equations. ◀

TODO: cite: refer to DSL impl chapter, and to literature (Czarnecki)

```
statemachine linefollower {
  event initialized;
  {bumper && debugOutput} event bumped;
  {sonar} event blocked;
  {sonar} event unblocked;
  initial state initializing {
    initialized [true] -> state running
  }
  {sonar} state paused {
            entry int16 i = 1;
            unblocked [true] -> state running
          }
  state running {
    {sonar} blocked [true] -> state paused
    {bumper} bumped [true] -> state crash
  }
  {bumper} state crash {
             <<transitions>>
           }
}
```

Figure 1.26: Code affected by a presence condition is highlighted in blue. The presence condition is rendered on the left of the affected code. It is a boolean expression over a set of (predefined) configuration parameters.

Example: Another example for declarative programming is the type system DSL used by MPS itself. Language developers specify

a set of type equations containing free type variables, among other things. A unification engine tries to solve the set of equations by assigning actual types to the free type variables so that the set of equations is consistent. We describe this approach in detail in section ◀

TODO: ref

Logic programming is another subparadigm of declarative programming where users specify logic clauses (facts and relations) as well as queries. A theorem prover tries to solve the queries.

The Prolog language works this way

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

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

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

Refrigerators: The cooling language comes with a simulator (Fig. 1.27) based on an interpreter where the behaviour of a cooling algorithm can be debugged. Events are explicitly created by the user, on a time scale that is compatible with the debugging process. ◀

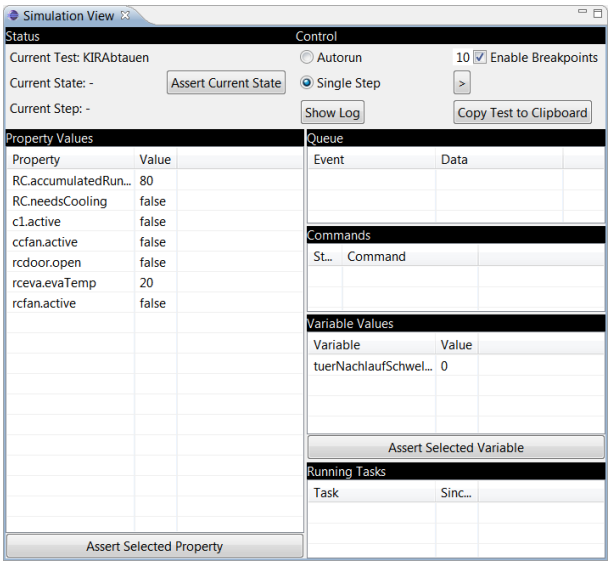

Figure 1.27: The simulator for the cooling language shows the state of the system (commands, event queue, value of hardware properties, variables and tasks). The program can be single-stepped. The user can change the value of variables or hardware properties as a means of interacting with the program.

Dataflow The dataflow paradigm is centered around variables with dependencies (relationships in terms of calculation rules) among them. As a variable changes, those variables that depend on the changing variable are recalculated. We know this approach mainly from two use cases. One is spreadsheets: cell formulas express dependencies to other cells. As the values in these other cells change, the dependent cells are updated. The other use case is data flow (or block) diagrams (Fig. 1.28, used in embedded software, ETL systems and enterprise messaging and complex event processing. There, the calculations are encapsulated in the blocks, and the lines represent dependencies — the output of one blocks "flows" into input slot of another block. There are three different execution modes:

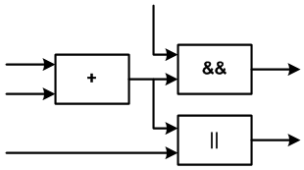

Figure 1.28: Graphical Notation for Flow

- The first one considers the data values as continuous signals. At the time one of the inputs changes, all dependent values are recalculated, the change triggers the recalculation, and the recalculation ripples through the dependency graph. This is the model used in spreadsheets.
- The second one considers the the data values quantized, unique messages. Only if a message is available for all inputs, a new output message is calculated. The recalculation synchronizes on the availability of a message at each input, and upon recalculation, these messages are consumed. This approach is often used in ETL and CEP systems.
- The third approach is time triggered. Once again, the inputs are

understood to be continuous signals, and a scheduler determines when a new calculation is performed. It also makes sure that the calculation "ripples through from left to right" in the correct order. This model is typically used in embedded systems.

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

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

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

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

As mentioned before, state-based behavior description is also interesting because it support model checking: a state chart is verified

A good introduction to model checking can be found in . We elaborate on model checking in section.

TODO: ref

against a set of specifications. The model checker either determines that the state chart conforms to the specifications or provides a counter example. Specifications express something about sequences of states such as: "it is not possible that two traffic lights show green at the same time" or "whenever a pedestrian presses the *request* button, the pedestrian lights eventually will show green".

1.7.3 Combinations

Many DSLs use combinations of various behavioural and structural paradigms described in this section[22]. A couple of combinations are very typical:

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

- a data flow language often uses a functional or imperative language to describe the calculation rules that express the dependencies between the variables (the contents of the boxes in data flow diagrams or of cells in spreadsheets)
- state machines use expressions as transition guard conditions, as well as typically an imperative language for expressing the actions that are executed as a state is entered or left, or when a transition is executed.
- reactive programming, where "black boxes" react to events, often use data flow or state-based programming to implement the behaviour that determines the reactions.
- in purely structural languages, for example, those for expressing components and their dependencies, a functional/expression language is often used to express pre- and postconditions for operations. A state-based language is often used for protocol state machines, which determines the valid order of incoming events or operation calls.

Some of the case studies used in this section of the book also use combinations of several paradigms.

Pension Plans: The pension language uses functional abstractions with mathematical symbols for the core actuary mathematics. A functional language with a normal textual syntax is used for the higher-level pension calculation rules. A spreadsheet/data flow language is used for expressing unit tests for pension rules. Various nesting levels of namespaces are used to organize the rules, the most important of which is the pension plan. A plan contains calculation rules as well as test cases for those rules. Pension plans can specialize other plans as a means of expressing variants. Rules in a sub-plan can override rules in the plan from which the

sub-plan inherits. Plans can be declared to be abstract, with abstract rules that have to be implemented in sub-plans. Rules are versioned over time, and the actual calculation formula is part of the version. Thus, a pension plan's behaviour can be made to be different for different points in time. ◄

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

1.8 Language Modularity

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

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

We have identified the following four composition strategies: referencing, extension, reuse and embedding. We distinguish them regarding fragment structure and language dependencies, as illustrated in Fig. 1.29. Fig. 1.30 shows the relationships between fragments and languages in these cases. We consider these two criteria to be essential for the following reasons.

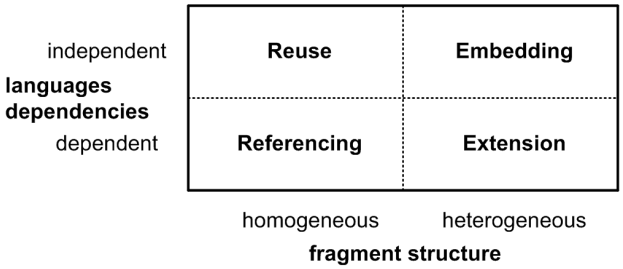

Figure 1.29: We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies.

Language dependencies capture whether a language has to be designed with knowledge about a particular composition partner in mind in order to be composable with that partner. It is desirable in many scenarios that languages be composable *without* previous knowledge

about all possible composition partners. *Fragment Structure* captures whether the two composed languages can be syntactically mixed. Since modular concrete syntax can be a challenge, this is not always that possible, though often desirable.

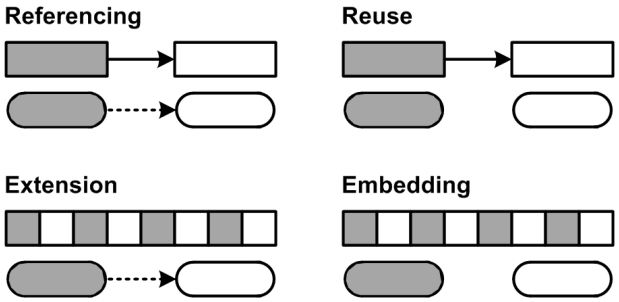

Figure 1.30: The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

1.8.1 *Language Referencing*

Language referencing (Fig. 1.31) enables *homogeneous* fragments with cross-references among them, using *dependent* languages.

A fragment f_2 depends on f_1. f_2 and f_1 are expressed with different languages l_2 and l_1. The referencing language l_2 depends on the referenced language l_1 because at least one concept in the l_2 references a concept from l_1. We call l_2 the *referencing* language, and l_1 the *referenced* language. While equations (2) and (3) continue to hold, (1) does not. Instead

$$\forall r \in Refs_{l_2} \mid lo(r.from) = l_2 \ \wedge \ lo(r.to) = (l_1 \vee l_2) \tag{1.8}$$

(we use $x = (a \vee b)$ as a shorthand for $x = a \vee x = b$).

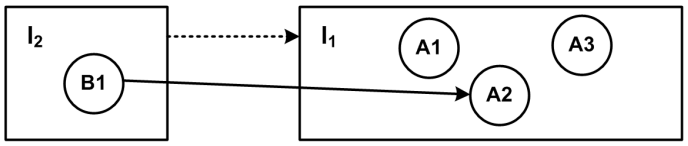

Figure 1.31: Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

Viewpoints As we have discussed before in Section 1.5, a domain D can be composed from different concerns. One way of dealing with this is to define a separate concern-specific DSLs, each addressing one or more of the domain's concerns. A program then consists of a set of concern-specific fragments, that relate to each other in a well-defined way using language referencing. The latter approach has the advantage that different stakeholders can modify "their" concern independent of others. It also allows reuse of the independent fragments and

languages with different referencing languages. The obvious drawback is that for tightly integrated concerns the separation into separate fragments can be a usability problem.

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

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

Progressive Refinement Progressive refinement, also introduced earlier (Section 1.5.3), also makes use of language referencing.

1.8.2 *Language Extension*

Language extension Fig. 1.31 enables *heterogeneous* fragments with *dependent* languages. A language l_2 extending l_1 adds additional language concepts to those of l_1. We call l_2 the *extending* language, and l_1 the *base* language. To allow the new concepts to be used in the context provided by l_1, some of them extend concepts in l_1. So, while l_1 remains independent, l_2 becomes dependent on l_1 since

$$\exists i \in Inh(l_2) \mid i.sub = l_2 \ \wedge \ i.super = l_1 \tag{1.9}$$

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

$$\forall e \in E_f \mid lo(e) = (l_1 \vee l_2) \tag{1.10}$$

In other words, $C_f \subset (C_{l_1} \cup C_{l_2})$, so f is *heterogeneous*. For heterogeneous fragments (3) does not hold anymore, since

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = (l_1 \vee l_2) \wedge lo(co(c.child)) = (l_1 \vee l_2) \quad (1.11)$$

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

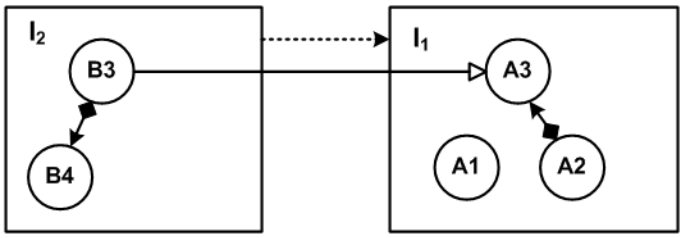

Figure 1.32: Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

Language extension fits well with the hierarchical domains introduced in Section 1.1.2: a language L_B for a domain D may extend a language L_A for D_{-1}. L_B contains concepts specific to D, making analysis and transformation of those concepts possible without pattern matching and semantics recovery. As explained in the introduction, the new concepts are often reified from the idioms and patterns used when using an L_A for D. Language semantics are typically defined by mapping the new abstractions to just these idioms (see Section 1.4) *inline*. This process, also known as *assimilation*, transforms a heterogeneous fragment (expressed in L_D and L_{D+1}) into a homogeneous fragment expressed only with L_D.

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

Embedded C: As an example consider embedded programming. The C programming language is typically used as the GPL for D_0in this case. Extensions for embedded programming include state machines, tasks or data types with physical units. Language extensions for the subdomain of real-time systems may include ways of specifying deterministic scheduling and worst-case execution time. For the avionics subdomain support for remote communication using some of the bus systems used in avionics could be added. ◄

TODO: cite Models Paper

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

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

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

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

TODO: cite

1.8.3 Language Reuse

Language reuse (Fig. 1.33) enables *homogenous* fragments with *independent* languages. Given are two independent languages l_2 and l_1 and two fragment f_2 and f_1. f_2 depends on f_1, so that

$$\exists r \in Refs_{f_2} \mid fo(r.from) = f_2 \wedge fo(r.to) = (f_1 \vee f_2) \tag{1.12}$$

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

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

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

$$\exists r \in Refs_{l_A} \mid lo(r.from) = l_A \wedge lo(r.to) = l_1 \tag{1.13}$$

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

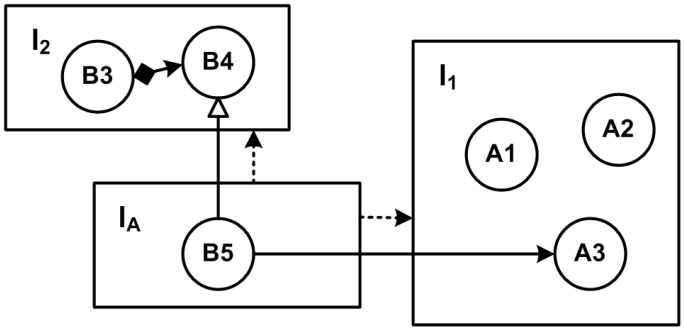

Figure 1.33: Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

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

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

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

1.8.4 Language Embedding

Language embedding (Fig. 1.34) enables *heterogeneous* fragments with *independent* languages. It is similar to reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now embed instances of

concepts from l_2 in a fragment f expressed with l_1, so

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge \\ lo(co(c.child)) = (l_1 \vee l_2) \quad (1.14)$$

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

$$\exists c \in Cdn_{l_A} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \quad (1.15)$$

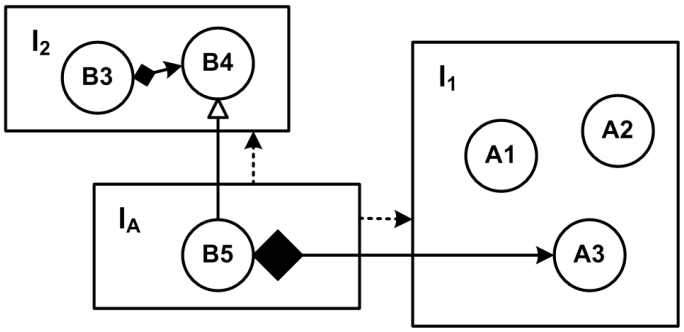

Figure 1.34: Embedding: l_1 and l_2 are independent languages. However, we still want to use them in the same fragment. To enable this, an adapter language l_A is added. It depends on both l_1 and l_2, and uses inheritance and composition to adapt l_1 to l_2 (this is the almost the same structure as in the case of reuse; the difference is that $B5$ now contains $A3$, instead of just referencing it.)

Embedding supports syntactic composition of independently developed languages. As an example, consider a state machine language that can be combined with any number of programming languages such as Java or C. If the state machine language is used together with Java, then the guard conditions used in the transitions should be Java expressions. If it is used with C, then the expressions should be C expressions. The two expression languages, or in fact, any other one, must be embeddable in the transitions. So the state machine language cannot depend on any particular expression language, and the expression languages of C or Java obviously cannot be designed with knowledge about the state machine language. Both have to remain independent, and have to be embedded using an adapter language.

When embedding a language, the embedded language must often be extended as well. In the above example, new kinds of expressions must be added to support referencing event parameters. These additional expressions will typically reside in the adapter language as well.

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

WebDSL: In order to support queries over persistent data, WebDSL embeds the Hibernate Query Language (HQL) such that HQL queries can be used as expressions. Queries can refer to entity

declarations in the program and to variables in the scope of the query. ◀

Pension Plans: The pension workbench DSL embeds a spreadsheet language for expressing unit tests for pension plan calculation rules. ◀

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

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

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

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

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

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

1.8.5 Implementation Challenges and Solutions

Syntax Referencing and Reuse keeps fragments homogeneous. Mixing of concrete syntax is not required. A reference between fragments

is usually simply an identifier and does not have its own internal structure for which a grammar would be required. The name resolution phase can then create the actual cross-reference between abstract syntax objects.

Refrigerators: The algorithm language contains cross-references into the hardware language. Those references are simple, dotted names (*a.b.c*). ◄

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

Extension and Embedding requires modular concrete syntax definitions because additional language elements must be "mixed" with programs written with the base language.

Embedded C: State machines are hosted in regular C programs. This works because the C language's `Module` construct contains a collection of `ModuleContents`, and the `StateMachine` concept extends the `ModuleContent` concept. This state machine language is designed specifically for being embedded into C, so it can access and extend the `ModuleContent` concept (Fig. 1.35). If the state machine language were reusable with any host language in addition to C, then an adapter language would provide a language concept that adapts C's `IModuleContent` to `StateMachine`, because `StateMachine` cannot directly extend `IModuleContent` — it does now depend on the C language. ◄

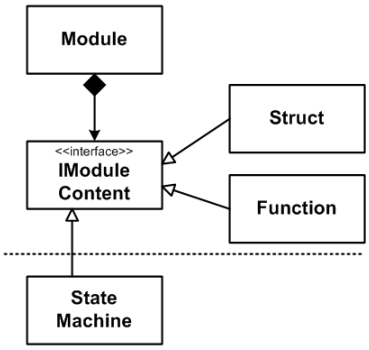

Figure 1.35: The core language (above the dotted line) defines an interface `IModuleContent`. Anything that should be hosted inside a `Module` has to implement this interface, typically from another language. `StateMachines` are an example.

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

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

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

Embedded C: A language extension provides types with physi-

cal units (as in 100 kg). Additional typing rules are needed to override the typing rules for C's basic operators (+, -, *, /, etc.). ◄

For reuse and embedding, the typing rules that affect the interplay between the two languages reside in the adapter language.

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

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

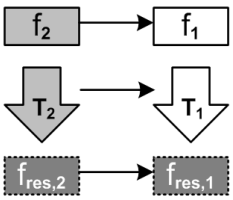

Figure 1.36: Referencing: Two separate, dependent, single-source transformations

Three cases have to be considered for referencing. The first one (Fig. 1.36) propagates the referencing structure to the target fragments. We call these two transformations single-sourced, since each of them only uses a single, homogeneous fragment as input and creates a single, homogeneous fragment, perhaps with references between them. Since the referencing language is created with the knowledge about the referenced language, the generator for the referencing language can be written with knowledge about the names of the elements that have to be referenced in the other fragment. If a generator for the referenced language already exists, it can be reused unchanged. The two generators basically share naming information.

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

The second case (Fig. 1.37) is a multi-sourced transformation that creates one single homogeneous fragment. This typically occurs if the referencing fragment is used to guide the transformation of the referenced fragment, for example by specifying target transformation strategies. In this case, a new transformation has to be written that takes the referencing fragment into account. The possibly existing generator for the referenced language cannot be reused as is.

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

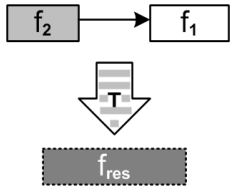

Figure 1.37: A single multi-sourced transformation.

The third case, an alternative to rewriting the generator, is the use of a preprocessing transformation (Fig. 1.38), that changes the referenced fragment in a way consistent with what the referenced fragment prescribes. The existing transformations for the referenced fragment can then be reused.

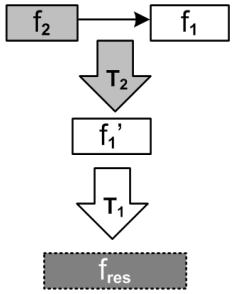

Figure 1.38: A preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment

As we have discussed above, language extensions are usually created by defining linguistic abstractions for common idioms of a domain D. A generator for the new language concepts can simply recreate those idioms when mapping L_D to L_{D-1}, a process called assimilation. In other words, transformations for language extensions map a heterogeneous fragment (containing L_{D-1} and L_D code) to a homogeneous fragment that contains only L_{D-1} code (Fig. 1.39). In some cases additional files may be generated, often configuration files.

Embedded C: State machines are generated down to a function that contains a switch/case statement, as well as enums for states and events. ◄

Sometimes a language extension requires rewriting transformations defined by the base language.

Embedded C: In the data-types-with-physical-units example, the language also provides range checking and overflow detection. So if two such quantities are added, the addition is transformed into a call to a special `add` function instead of using the regular plus operator. This function performs overflow checking and addition. ◄

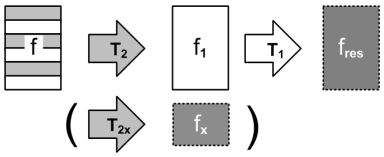

Figure 1.39: Extension: transformation usually happens by assimiliation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often some configuration files.

Language Extension introduces the risk of semantic interactions. The transformations associated with several independently developed extensions of the same base language may interact with each other.

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

Interactions may also be more subtle and affect memory usage or execution performance. Note that this problem is not specific to languages, it can occur whenever several independent extensions of a base concept can be used together, ad hoc. To avoid the problem, transformations should be built in a way so that they do not "consume scarce resources" such as inheritance links. A more thorough discussion of the problem of semantic interactions is beyond the scope of this paper, and we refer to for details.

TODO: cite

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

There are three cases to consider. The first one, illustrated in Fig. 1.40, describes the case where there is an existing transformation for the reused fragment and an existing transformation for the context fragment — the latter being written with the knowledge that later extension will be necessary. In this case, the generator for the adapter language may "fill in the holes" left by the reusable generator for the referencing language For example, the generator of the context language may generate a class with abstract methods; the adapter may generate a subclass and implement these abstract methods.

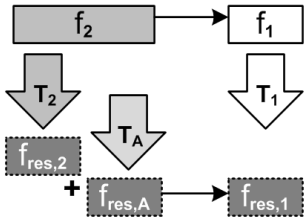

Figure 1.40: Reuse: Reuse of existing transformations for both fragments plus generation of adapter code

In the second case, Fig. 1.41, the existing generator for the reused fragment has to be enhanced with transformation code specific to the context language. In this case, a mechanism for composing transformations is needed.

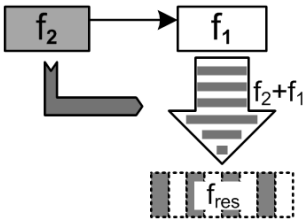

Figure 1.41: Reuse: composing transformations

The third case, Fig. 1.42, leaves composition to the target languages. We generate three different independent, homogeneous fragments, and a some kind of weaver composes them into one final, heterogeneous artifact. Often, the weaving specification is the intermediate result generated from the adapter language. An example implementation could use AspectJ.

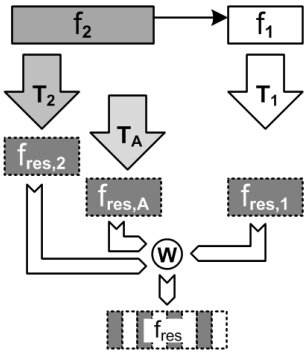

Figure 1.42: Reuse: generating separate artifacts plus a weaving specification

Embedding An embeddable language may not come with its own generator, since, at the time of implementing the embeddable language, one cannot know what to generate. In that case, when embedding the language, a suitable generator has to be developed. It will typically

either generate host language code (similar to generators in the case of language extension) or directly generate to the same target language that is generated to by the host language.

If the embeddable language comes with a generator that transforms to the same target language as the embedding language, then the generator for the adapter language can coordinate the two, and make sure a single, consistent fragment is generated. Fig. 1.43 illustrates this case.

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

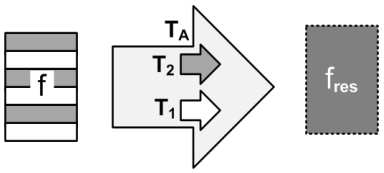

Figure 1.43: In transforming embedded languages, a new transformation has to be written if the embedded language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the emebedded languages.

1.9 *Concrete Syntax*

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

There are a couple of major classes for DSL concrete syntax[23]: *Textual* DSLs use traditional ASCII or Unicode text. They basically look and feel like traditional programming languages. *Graphical* DSLs use graphical shapes. An important subgroup is represented by those that use box-and-line diagrams that look and feel like UML class diagrams or state machines. *Symbolic* DSLs are textual DSLs with an extended set of symbols, such as fraction bars, mathematical symbols or subscript and superscript. However, there are more options for graphical notations, such as those illustrated by UML timing diagrams or sequence diagrams. *Tables and Matrices* are a powerful way to represent certain kinds of data and can play an important part for DSLs.

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

When to use which form We do not want to make this section a complete discussion between graphical and textual DSLs — a discussion, that is often heavily biased by previous experience, prejudice and tool capabilities. Here are some rules of thumb. Purely textual DSLs in-

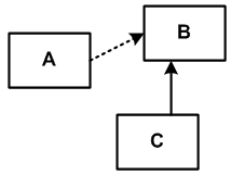

Figure 1.44: Graphical Notation for Relationships

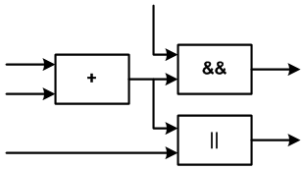

Figure 1.45: Graphical Notation for Flow

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

The Fortress programming language is close to this.

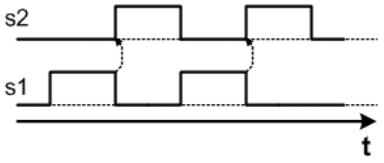

Figure 1.46: Graphical Notation for Causality and Timing

tegrate very well with existing development infrastructures, making their adoption relatively easy. They are very well suited for detailed descriptions, anything that is algorithmic or generally resembles (traditional) program source code. Symbolic notations can be considered "better textual", and lend themselves to domains that make heavy use of symbols and special notations. tables are very useful for collections of similarly structured data items, or for expressing how two independent dimensions of data relate. Finally, graphical notations are very good for describing relationships (Fig. 1.44), flow (Fig. 1.45) or timing and causal relationships (Fig. 1.46).

Pension Plans: The pension DSL uses mathematical symbols and notations to express insurance mathematics (Fig. 1.47). A table notation is embedded to express unit tests for the pension plan calculation rules. A graphical projection shows dependencies and specialization relationships between plans. ◄

Embedded C: The core DSLs use a textual notation with some tabular enhancements, for example, for decision tables (Fig. 1.48). However, as MPS' capability for handling graphical notations will increase, we will represent state machines as diagrams. ◄

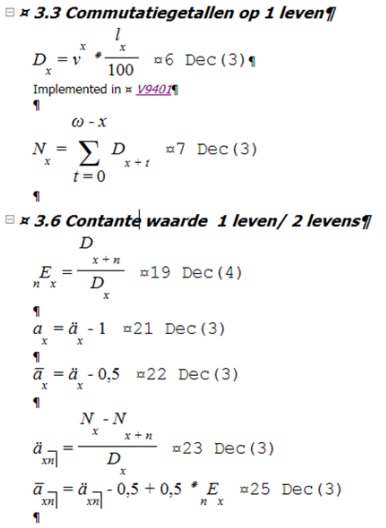

Figure 1.47: Mathematical notations used to express insurance math in the pension workbench.

```
c/s interface Decider {
  int decide(int x, int y) pre
}

component AComp extends nothing {
  ports:
    provides Decider decider
  contents:
    int decide(int x, int y) <- op decider.decide {
      return int, 0
```

	x == 0	x > 0
y == 0	0	1
y > 0	1	2

```
    }
}
```

Figure 1.48: Decision tables use a tabular notation. It is embedded seemlessly into a C program.

Selection of concrete syntax is simple for domain user DSLs if there is an established notation in the domain. The challenge then is to replicate it as closely as possible with the DSL, while cleaning up possible inconsistencies in the notation (since it had not been used formally before). I like to use the term "strongly typed word" in this case.[24]

For DSLs targeted at developers, a textual notation is usually a good starting point, since developers are used working with text, and very productive. Tree views, and specific visualizations are often useful, but for editing, a textual notation is a good starting point. It also integrates

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

well with existing development infrastructures.

Embedded C: This was the original reason for developing this system as a set of extensions to C. C is the baseline for embedded systems, and everybody is familiar with it. A textual notation is useful for many concerns in embedded systems. Note that several languages create visualizations on the fly, for example for module dependencies, component dependencies and component instance wirings. The graphviz tool is used here since it provides decent auto-layout. ◄

TODO: ref

There are very few DSLs where a purely graphical notation makes sense. In most cases, textual languages are embedded in the diagrams or tables: state machines have expressions embedded the guards and statements in the actions (); component diagrams use text for specifications of operations in interfaces, maybe even with expressions for preconditions; block diagrams use a textual syntax for the implementation/parametrization of the blocks (Fig. 1.49); tables may embed textual notations in the cells (Fig. 1.50). .

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

TODO: screenshots from some itemis tools that combine textual and graphical

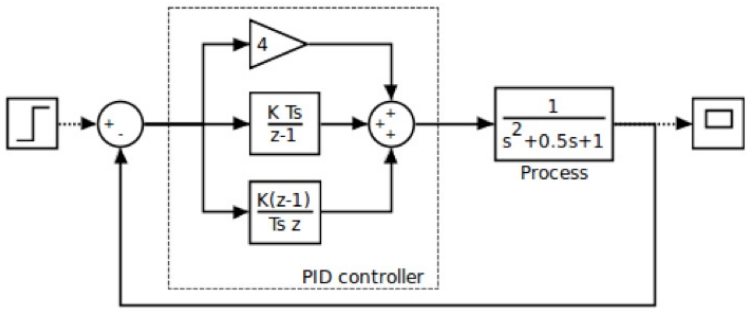

Figure 1.49: Block diagrams built with the Yakindu modeling tools. A textual DSL is used to implement the behaviour in the blocks. While the textual DSL is not technically integrated with the graphical notation (separate viewponts), semantic integration is provided.

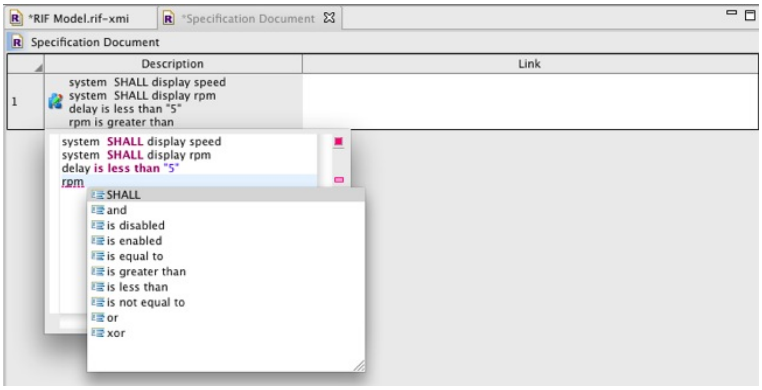

Figure 1.50: The Yakindu Requirements tools integrates a textual DSL for formal requirements specification. An Xtext DSL is syntactically integrated into a table view. Semantic integration is provided as well.

Also note that initially, domain users prefer a graphical notation, because of the perception that things that are described graphically are simple(r) to comprehend Not really. However, what is most important

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

regarding comprehensibility is the alignment of the domain concepts with the abstractions in the language. A well-designed textual notation can go a long way. Also, textual languages are more productive once the learning curve has been overcome.

Multiple Notations For projectional editors it is possible to define several notations for the same abstract syntax. By changing the projection rules, existing programs can be shown in a different way. This removes some of the burden of getting it right initially, because the notation can be adapted after the fact. In general, for the concrete syntax of a DSL writability is often more important than readability, because additional read-only representations can always be derived automatically.

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

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

Relationship to Hierarchical Domains Domains at low D are most likely best expressed with a textual or symbolic concrete syntax. Obvious examples include programming languages at D_0. Mathematical expressions, which are also very dense and algorithmic, use a symbolic notation. As we progress to higher Ds, the concepts become more and more abstract, and as state machines and block diagrams illustrate, graphical notations become useful. However, these two notations are also a good example of language embedding since both of them require expressions: state machines in guards, and block diagrams as the implementation of blocks. Reusable expression languages should be embedded into the graphical notations. In case this is not supported by the tool, viewpoints may be an option. One viewpoint could use a graphical notation to define coarse-grained structures, and a second viewpoint uses a textual notation to provide "implementation details" for the structures defined by the graphical viewpoint[25].

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

[25] Not every tool can support every (combination of) form of concrete syntax, so this aspect is limited by the tool, or drives tool selection.
TODO: The above para needs to be refactored to the Yakindu SM and Data FLow examples fit in well.

2
Process Issues

TODO: HERE

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

2.1 DSL Development

2.1.1 Requirements for the Language

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

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

In case of business domain DSLs, you can often mine the existing (tacit) knowledge of domain experts. In domains like insurance, science or logistics, domain experts are absolutely capable of precisely expressing domain knowledge. They do it all the time, often using Excel or Word. They often have a "language" to express domain concerns, although it is usually not formal, and there's no tool support. In this context, your job is to provide formality and tooling. Similar to domain knowledge, other domain artifacts can also be exploited: for example, hardware structures or device features are good candidates for abstractions in the respective domains.

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

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

However, in the remaining third case, however, we are not so lucky. If no domain knowledge is easily available, we have to do an actual domain analysis, digging our way through requirements, stakeholder "war stories" and existing applications. Co-evolving language and concepts (see below) is a successful technique especially in this case.

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

2.1.2 *Iterative Development*

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

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

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

It is important that the language approaches some kind of stable state over time (Fig. 2.1). As you iterate, you will encounter the following situation: domain experts express requirements that may sound inconsistent. You add all kinds of exceptions and corner cases to the language. You language grows in size and complexity. After a num-

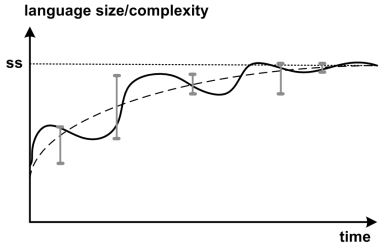

Figure 2.1: Iterating towards a stable language over time

ber of these exceptions and corner cases, ideally the language designer will spot the systematic nature behind these and refactor the language to reflect this deeper understanding of the domain. Language size and complexity is reduced. Over time, the amplitude of these changes in language size and complexity (error bars in Fig. 2.1) should become smaller, and the language size and complexity should approach a stable level (*ss* in Fig. 2.1).

2.1.3 Co-evolve concepts and language

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

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

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

2.1.4 Let people do what they are good at

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

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

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

There's also a flip side here: you have to make sure you actually do have people on your team who are good at language design, know

about the domain and understand target platforms. Otherwise the MD* approach will not deliver on its promises.

2.1.5 Domain Users vs. Domain Experts

When building business DSLs, people from the domain can play two different roles. They can participate in the domain analysis and the definition of the DSL itself. On the other hand, they can use the DSL to express specific domain knowledge.

It is useful to distinguish these two roles explicitly. The first role (language definition) must be filled by a domain expert. These are people who have typically been working in the domain for a long time, maybe in different roles, who have a deep understanding of the relevant concepts and they are able to express them precisely, and maybe formally.

The second group of people are the domain users. They are of course familiar with the domain, but they are typically not as experienced as the domain experts

This distinction is relevant because you typically work with the domain experts when defining the language, but you want the domain users to actually work with the language. If the experts are too far ahead of the users, the users might not be able to "follow" along, and you will not be able to roll out the language to the actual target audience.

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

2.1.6 DSL as a Product

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

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

2.1.7 *Documentation is still necessary*

Building DSLs and model processors is not enough to make the approach successful. You have to communicate to the users how the DSL and the processors work. Specifically, here's what you have to document: the language structure and syntax, how to use the editors and the generators, how and where to write manual code and how to integrate it as well as platform/framework decisions (if applicable).

Please keep in mind that there are other media than paper. Screencasts, videos that show flipchart discussions, or even a regular podcast that talks about how the tools change are good choices, too.

Also keep in mind that hardly anybody reads reference documentation. If you want to be successful, make sure the majority of your documentation is example-driven or task-based.

2.2 *Using DSLs*

2.2.1 *Reviews*

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

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

Reviews are easier on DSL level than on code level. Since DSL programs are more concise than their equivalent specification in GPL code, reviews become more efficient.

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

2.2.2 *Compatible Organization*

Done right, MD* requires a lot of cross-project work. In many settings the same metaware will be used in several projects or contexts. While this is of course a big plus, it also requires, that the organization is able to organize, staff, schedule and pay for cross-cutting work. A strictly

project-focused organization has a very hard time finding resources for these kinds of activities. MD* is very hard to do effectively in such environments.

Make sure that the organizational structure, and the way project cost is handled, is compatible with cross-cutting activities. You might want to take a look at the Open Source communities to get inspirations of how to do this.

2.2.3 *Domain Users Programming?*

Domain users aren't programmers. But they are still able to formally and precisely describe domain knowledge. Can they actually do this alone?

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

Initially, you might even want to reduce your aspirations to the point where the developer does the DSL coding based on discussions with domain users, but then showing them the resulting model and asking confirming or disproving questions about it. Putting knowledge into formal models helps you point out decisions that need to be made, or language extensions that might be necessary.

Executing the program, by generating code or running some kind simulator can also help domain users understand better what has been expressed with the DSL.

If you're not able to teach a business domain DSL to the domain users, it might not necessarily be the domain users' fault. Maybe your language isn't really suitable to the domain. If you encounter this problem, take it as a warning sign and take a close look at your language.

TODO: put somewhere?