*4*

## Design Dimensions

This chapter has been written in association with Eelco Visser of TU Delft. You can contact him via **e.visser@tudelft.nl**.

*DSLs are powerful tools for software engineering, because they can be tailor-made for a specific class of problems. However, because of the large degree of freedom in designing DSLs, and because they are supposed to cover the intended domain, consistently, and at the right abstraction level, DSL design is also hard. In this chapter we present a framework for describing and characterizing domain specific languages. We identify seven design dimensions that span the space within which DSLs are designed: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax.*

We illustrate the design alternatives along each of these dimensions with examples from our case studies. The dimensions provide a vocabulary for describing and comparing the design of existing DSLs, and help guide the design of new ones. We also describe drivers, or forces, that lead to using one design alternative over another. This chapter is not a complete methodology. It does not present a recipe that guarantees a great DSL if followed. I don’t believe in methodologies, because they pretend precision where there isn’t any. Building a DSL is a craft. This means that, while there are certain established approaches and conventions, building a good DSL also requires experience and practice.

### 4.1 Expressivity

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

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

|  |  |
| --- | --- |
| to indicate the size of program *p* as encoded in language *L*2. The essence is the assumption that, within one language, more complex programs will require larger encodings. We also assume that *pL* 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 in domain D* than a language *L*2 (*L*1 ≺*D L*2),  if for each *p* ∈ *PD* ∩ *PL*1 ∩ *PL*2, |*pL*1| *<* |*pL*2|.  A weaker but more realistic version of this statement requires that a language is *mostly* more expressive, but may not be in some obscure special cases: DSLs may optimize for the common case and may require code written in a more general lan- |  |
| guage to cover the corner cases3. |  |

While it is always possible to produce short but incomprehensible programs, in general shorter programs require less effort to read and write than longer programs, and are therefore be more efficient in software engineering. We will thus assume that, all other things being equal, shorter programs are preferable over longer programs.1. We use the notation |*pL*|,

Compared to GPLs, DSLs (and the programs expressed with them) are more *abstract*: they avoid describing details that are irrelevant to the model purpose. The execution engine then fills in the missing details to make the program executable on a given target platform, based on knowledge about the domain encoded in the execution engine. Good DSLs are also *declarative*: they provide linguistic abstractions for relevant domain

concepts that allow processors to "understand" the domain semantics without sophisticated analysis of the code. Linguistic abstraction means that a language contains concepts for the abstractions relevant in the domain. We discuss this in more detail below.

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

|  |  |
| --- | --- |
| learning the domain itself, and learning the syntax of the language. For people who understand 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 understand the domain, or would have to learn the domain even if no DSL were used to express programs in it: 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 DSL5. In conclusion, the |  |

DSLs have the advantage of being more expressive than GPLs in the domain they are built for. But there is also a disadvantage: before being able to write these concise programs, users have to learn the language4. This task can be separated into

learning overhead of DSLs is usually not a huge problem in practice.

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

The close alignment between a domain and the DSL can also be exploited during the construction of the DSL. While it is not a good idea to start building a DSL for a domain about which

we don’t know much, the process of building the DSL can help deepen the understanding about a domain. The domain has to be scoped, fully explored and systematically structured to be able to build a language.

**Refrigerators:** Building the cooling DSL has helped the thermodynamicists and software developers to understand the details of the domain, its degrees of freedom and the variability in refrigerator hardware and cooling algorithms in a much more structured and thorough way than before. Also, the architecture of the 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. J

#### 4.1.1 Expressivity and the Domain Hierarchy

In the section on expressivity above we compare arbitrary languages. An important idea behind domain-specific languages is that progressive specialization of the domain enables progressively more specialized and expressive languages. Programs for domain *Dn* ⊂ *Dn*−1 expressed in a language *LDn*−1 typically use a set of characteristic idioms and patterns. A language for *Dn* can provide linguistic abstractions for those idioms or patterns, which makes their expression much more concise and their analysis and translation less complex.

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

|  |  |
| --- | --- |
| **WebDSL:** WebDSL entity declarations abstract over the |  |
| boilerplate code required by the Hibernate6 framework for annotating Java classes with object-relational mapping annotations. This reduces code size by an order of magni- |  |
| tude 7. J |  |
| *4.1.2 Linguistic versus In-Language Abstraction* |  |

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

*Linguistic Abstraction* A specific domain concept can be modeled with the help of existing abstractions, or one can introduce a *new* abstraction for that concept. If we do the latter, we use *linguistic* abstraction. By making the concepts of *D* first-class members of a language *LD*, i.e. by defining linguistic abstractions for these concepts, they can be uniquely identified in a *D* program and their structure and semantics is well defined. No semantically relevant8 idioms or patterns are re-

quired to express interesting programs in *D*. Consider these two examples of loops in a Java-like language:

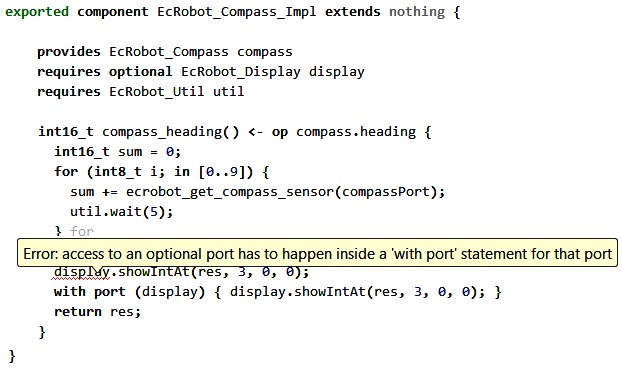
|  |  |
| --- | --- |
| **int**[] arr = ... | **int**[] arr = ... |
| **for** (**int** i=0; i<arr.size(); i++) { | OrderedList<**int**> l = ... |
| sum += arr[i]; | **for** (**int** i=0; i<arr.size(); i++) { |
| } | l.add( arr[i] );  } |

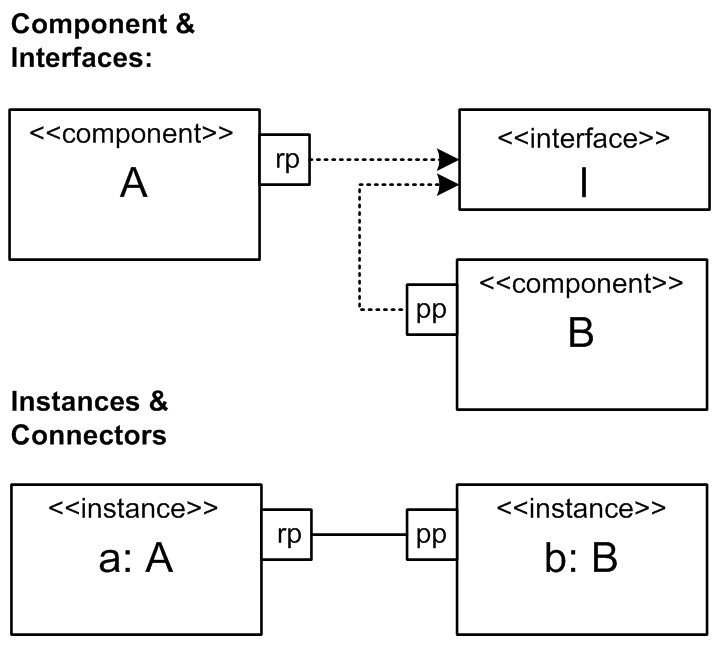
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| The loop in the left-hand example can be parallelized, since the order of summing up the array elements is irrelevant. The right-hand one cannot, since the order of the elements in the **OrderedList** class *is* relevant. A transformation engine that translates and optimizes these programs must perform (sophisticated, and sometimes impossible) program analysis to determine that the left-hand loop example can indeed be parallelized. The following alternative expression of the same behavior uses better linguistic abstractions, because it is clear without analysis that the first loop can be parallelized and the second cannot:   |  |  | | --- | --- | | **for** (**int** i in arr) { | seqfor (**int** i in arr) { | | sum += i; | l.add( arr[i] ); | | } | } |   The property of a language *LD* of having first-class concepts for abstractions relevant in *D* is often called *declarativeness*: no sophisticated pattern matching or program flow analysis is necessary to capture the semantics of a program (relative to the purpose) and treat it correspondingly. The decision can simply be based on the language concept used (**for** versus **seqfor**)[[1]](#footnote-1).  **mbeddr 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 switchcase statements or indexed arrays that constitute the implementation of the state machine10. J

1. This approach assumes that the generator works correctly – we’ll discuss this problem in Section 4.3 on semantics.

**mbeddr C:** Another good example is optional ports in components. Components (see Fig. 20.6) define required ports that specify the interfaces they *use*. For each component instance, each required port is connected to a provided port of another instance (that has a compatible interface). Required ports may be optional11, so for a given instance, an optional port may be connected or not. Invoking an operation on an unconnected required port would result in an error, so this has to be prevented. This can be done by enclosing the invocation on a required port in an **if** statement, checking whether the port is connected. However, an **if** statement can contain any arbitrary Boolean expression as its condition (e.g., **if (isConnected(rp) || somethingRandom()) { port.doSomething(); }**). So checking *statically* that the invocation only happens if the port is connected is impossible. A better solution based on linguistic abstraction is to introduce a new language concept that checks for a connected port directly: **with port (rp) { rp.doSomething(); }**. The **with port** statement doesn’t use an expression as its argument, but only a reference to an optional required port (Fig. 4.2). In this way the IDE can check that an invocation on a required optional port **rp** is only done inside of a **with port** statement referencing that same port. J





.

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 (over-specification). Overspecification is usually bad because it limits the degrees of freedom available to a transformation engine. In the example with the parallelizable loops, the first loop is over-specified: it expresses ordering of the operations, although this is (most probably) not intended by the person who wrote the code.

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

|  |  |
| --- | --- |
| abstraction mechanisms as long as the abstractions created don’t require special treatment by analysis or processing tools – at which point they should be refactored into linguistic abstractions. An example of such special treatment would be if the compiler of the above example language knew that the **OrderedList** library class is actually ordered, and that, consequently, the respective loop cannot be parallelized. Another example of special treatment can be constructed in the context of the optional port example. If we had solved the problem by having a library function **isConnected(port)**, we could enforce a call on an optional port to be surrounded by an **if (isConnected (port))** *without any other expression* in the condition. In this case, the static analyzer would have to treat **isConnected** spe- |  |
| cially12. In-language abstraction can, as the name suggests, |  |

*In-Language Abstraction* Conciseness can also be achieved by a language that provides facilities to allow users to define new (non-linguistic) abstractions in programs. Well-known GPL concepts for building new abstractions include procedures, classes, or functions and higher-order functions, generics, traits and monads. It is *not* a sign of a bad DSL if it has in-language

provide *abstraction*, but it cannot provide *declarativeness*: a model processor has to "understand" what the user wanted to express by building the in-language abstraction, in order to be able to act on it.

**Refrigerators:** The language does not support the construction of new abstractions since its user community consists of 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 from the end users which had to be integrated directly into the language. J

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

**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. J

*Standard Library* If a language provides support for inlang- uage abstraction, these facilities can be used by the *language developer* to provide collections of domain specific abstractions to language users. Instead of adding language features, a standard library is deployed along with the language to all its users. It contains abstractions relevant to the domain,

This approach is of course well known

expressed as in-language abstractions. This approach keeps the language itself small, and allows subsequent extensions of the library without changing the language definition and processing tools.

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

|  |
| --- |
| **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: **int readonly** |

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

|  |
| --- |
| **property compartment**::isCooling: **bool readonly**  } |

*Comparing Linguistic and In-Language Abstraction* A language that contains linguistic abstractions for all relevant domain concepts is simple to process; the transformation rules can be tied to the identities of the language concepts. It also makes the language suitable for domain experts, because relevant domain concepts have a direct representation in the language. Code completion can provide specific and meaningful support for "exploring" how a program should be written. However, using linguistic abstractions extensively requires that the relevant abstractions be known in advance, or frequent evolution of the language is necessary. It can also lead to languages that feel large, bloated or inelegant. In-language abstraction is more flexible, because users can build just those abstractions they actually 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, in which one set of users develops the abstractions to be used by another set of developers. This is especially useful if the same language is to be used for several, related, projects or user groups. Each can build their own set of abstractions in the library.

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

#### 4.1.3 Language Evolution Support

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

Whether model migration is a challenge or not depends on the tooling. There are tools that make model evolution very smooth, but many environments don’t. Consider this when

|  |  |
| --- | --- |
| deciding on the tooling you want to use!  It is always a good idea to minimize those changes to a DSL that break existing models13. Backward compatibility and deprecation are techniques well worth keeping in mind when working with DSLs. For example, instead of just changing an existing concept in an incompatible way, you may add a new concept in addition to the old one, along with deprecation of the old one and a migration script or wizard. Note that you might be able to instrument your model processor to collect statistics on whether deprecated language features continue to be used. Once no more instances show up in models, you can safely remove the deprecated language feature. |  |
| 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 versions14. Alternatively, the |  |

set of all models can be migrated to a newer version using a script provided by the language developers. In cases where the set of users, and the DSL programs, are not easily accessible, much more effort must be put into maintaining backward compatibility, and the need for coordinated evolution should be kept minimal[[2]](#footnote-2).

#### 4.1.4 Precision versus Algorithm

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

In this case, a DSL has to be defined that abstracts far enough to hide these algorithmic details. Alternatively, you can create an incomplete language (Section 4.5) and have developers fill in the algorithmic details in GPL code. One way to do this is to provide a set of predefined behaviors (in some kind of library) which are then just parametrized or configured by the users.

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

#### 4.1.5 Configuration Languages

Configuration languages are purely declarative. They consist of a well-defined set of configuration parameters and constraints among them. "Writing programs" boils down to setting values for these parameters. In many cases, the parameters are Booleans, in which case a program is basically a selection of a subset of the configuration switches. Feature models constitute a well-known configuration language. We discuss configuration languages in more detail in the chapter on DSLs and Product Line Engineering (Section 21).

#### 4.1.6 Platform Influence

In theory, the design of the abstractions used in a language should be independent of the execution engine and the platform. However, this is not always the case[[3]](#footnote-3). There are two

reasons why the platform may influence the language.

*Runtime Efficiency* In most systems, the resulting system has to execute in a reasonably efficient way. Efficiency can mean performance, scalability, as well as resource consumption (memory, disk space, network bandwidth). Depending on the semantic gap between the platform and the language, building efficient code generators can be a lot of work (we discuss this in

|  |  |
| --- | --- |
|  |  |
| **mbeddr C:** The language does not support dynamically growing lists, because it is hard to implement them in an efficient way considering we are targeting embedded software. Dynamic allocation of memory is often not allowed, and even if it were, the necessary copying of existing list data into a new, bigger buffer is too expensive for practical use. The incurred overhead is also *not* obvious to the language user (he just increases list size or adds another element that triggers list growth), making it all the more dangerous. J  **mbeddr C:** Another example includes floating point arithmetic. If the target platform has no floating point unit (FPU), floating point arithmetic is expensive to emulate. We had to build the language in a way that could prevent the use of **float** and **double** types if the target platform had no FPU. J  *Platform Limitations* The platform may have limitations regarding the size of data structures, the memory or disk space, or the bandwidth of the network, that limit or otherwise influence language design.  **Refrigerators:** In the cooling language we had to introduce time units (seconds, minutes, hours) into the DSL after we’d noticed that the time periods relevant for cooling algorithms were so diverse that no single unit could fit all necessary values into the available integer types. If we had used only seconds, the days or months periods would not fit into the available **int**s. Using only hours or days obviously would not let us express the short periods without using fractions of floating point data types. So the language now has the ability to express periods, as in **3s** |  |
| or **30d**. J  *4.2 Coverage* |  |

some detail in the section on semantics (Section 4.3)). Instead of building the necessary optimizers, you can also change the language to use abstractions that make global optimizations simpler to build. 17.

A language *L* always defines a domain *D* such that *PD* = *PL*. Let’s call this domain *DL*, i.e. the domain determined by *L*.

This does not work the other way around: given a (deductively defined) domain *D*, there is not necessarily a language that *fully covers* that domain unless we revert to a universal language at a *D*0 (cf. the hierarchical structure of domains and languages).

A language *L fully covers* domain *D* if for each program *p* relevant to the domain *PD* a program *pL* can be written in *L*. In other words, *PD* ⊆ *PL*.

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

*CD*(*L*) = *number of PD programs expressable by L*

*number of programs in domain D*

At first glance, an ideal DSL will cover all of its domain (*CD*(*L*) 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 will evolve and grow, and the language has to be continuously evolved to retain full coverage.

|  |  |
| --- | --- |
|  |  |
| In addition to the evolution-related reason given above, there are two reasons for a DSL *not* to cover all of its *own* domain *D*. First, the language may be deficient and need 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 do- |  |
| main18. In this case, the remaining parts of *D* may have to |  |
| be expressed with code written in *D*−1 (see also Section 4.5). 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. J

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

**mbeddr 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 implemented 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. J

### 4.3 Semantics and Execution

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

Using a function *OB* that defines this observable behavior, we can define the semantics of a program *pLD* by mapping it to a program *q* in a language for *D*−1 that has the same observable behavior:

*semantics*(*pLD* ) := *qLD*−1 *where OB*(*pLD* ) == *OB*(*qLD*−1)

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

The technical implementation of the mapping to *D*−1 can be provided in two different ways: a DSL program can literally be transformed into a program in an *LD*−1, or an interpreter can be written in *LD*−1 or *LD*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.

#### 4.3.1 Static Semantics/Validation

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

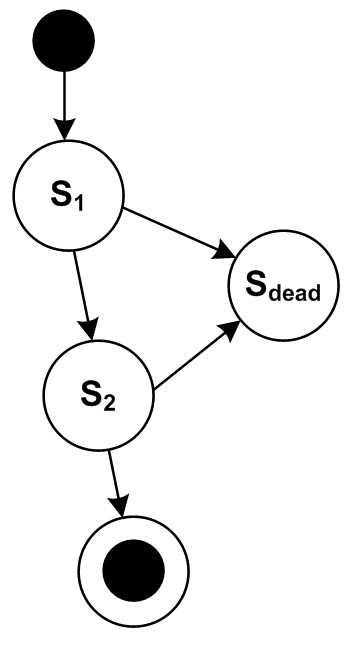
*Constraints* Constraints are simply Boolean expressions that check some property of a model. For example, one might verify that the names of a set of attributes of some entity are unique. For a model to be statically correct, all constraints have to evaluate to **true**. Constraint checking should only be performed for a model that is structurally/syntactically correct19.

**mbeddr C:** One driver in selecting the linguistic abstractions that go into a DSL is the ability to easily implement meaningful constraints. For example, in the state machine extension it is trivial to find states that have no outgoing transitions (dead end, Fig. 4.3). In a functional language, such a constraint could be written in the way shown in the code below. J

|  |
| --- |
| states  .select(s|!s.isInstanceOf(StopState))  .select(s|s.transitions.size == 0) |

When defining languages and transformations, developers often have a set of constraints in their mind that 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 transformation is used in which a model expressed in *L*3 is transformed into a model expressed in *L*2, which is then in turn transformed into a program ex-



|  |
| --- |
| pressed in *L*120. In this case it is important that *every* valid |

program in *L*3 leads to a valid program in *L*2. If the processing of *L*2 fails with an error message using abstractions from *L*2 (e.g., compiler errors), users of *L*3 will not be able to understand them; they may have never seen the programs generated in *L*2. Again, automated testing is the way to address this issue.

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

*Type Systems* Type systems are a special kind of constraint.

Consider the example of **var int x = 2 \* someFunction( sqrt(2));**. The type system constraint may check that the type of the variable is the same or a supertype of the type of the initialization expression. However, establishing the type of the init expression is non-trivial, since it can be an arbitrarily complex expression. A type system is a formalism or framework for defining the rules to establish the types of arbitrary expressions, as well as type checking constraints. It is a form of constraint checking. We cover the implementation of type systems in Part III of the book (Section 10).

When designing constraints and type systems 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.

**mbeddr C:** For variables, the type has to be specified explicitly. A type specification expresses the intent that this variable be, for example, of type **int**. Alternatively, a type system could be built to automatically derive the type of the variable declaration based on the type of the **init** expression, an approach known as *type inference*. This would

|  |
| --- |
| allow the following code: **var x = 2 \* someFunction( sqrt(2));**. Since no type is explicitly specified, the type system will infer the type of **x** to be the type calculated for the init expression. J  **mbeddr 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 specific ways of writing actions as invalid, because they cannot be handled by the model checker. An alternative approach could check a state machine for whether these "unverifiable" ways of writing actions are used, and if so, mark the state machine as not verifiable. J  **Pension Plans:** Pension plans can inherit from another plan (called the base plan). If a pension calculation rule overrides a rule in the base plan, then the overriding rule has to be marked as *overrides*. In this way, if the rule in the base plan is removed or renamed, validation of the sub-plan will report an error. An alternative design would simply infer the fact that a rule overrides another one if they have the same name and signature. J  Note how in all three cases the constraint checking is done in two steps. First we declare an *intent* (variable is intended to be **int**, this state machine is intended to be verifiable, a rule is intended to override another one). We then *check* whether the program conforms to this intention. The alternative approach would infer the fact from the program (the variable’s type is whatever the expression’s type evaluates to, state machines are verifiable if the "forbidden" features aren’t used, rules override another rule 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 trade-offs. The specification/conformance approach requires more code to be written, but results in more meaningful and specific error messages. A message can express that fact that one part of a program does not conform to a specification made by another part of the program[[4]](#footnote-4). The derivation/con- |

sistency 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.

The specification/conformance approach can also be used to "anchor" the constraint checker, because a fixed fact about the program is explicitly given instead of having to be derived from a (possibly large) part of the program. This decouples models and can increase scalability. Consider the following example. A program contains a function call, and the type checker needs to check the typing for this call. To do so, it has to determine the type of the called function. Assume this function does *not* specify the return type explicitly, instead it is inferred from the returned expressions. These expressions may be calls to other functions, so the process repeats. In the worst case, a whole chain of function calls must be followed in this way to calculate the type of the function initially called by your program. Notice that this requires accessing all the downstream programs, so these all have to be loaded and type checked! In large systems,

this can lead to serious performance and scalability issues22. If, type of the called function were given explicitly, nodownstream models need to be accessed or loaded.

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

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

**mbeddr C:** Another example concerns the use of floating point types. Some target devices may not have floating point units (FPUs), which means that floating point

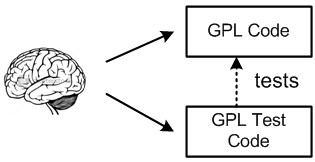
types (**float**, **double**) cannot be used in programs that should be deployed on such a target device. So, as the user changes the target device in the build configuration, additionawritel constraints are checked that report floating point types as errors if the target has no FPU. J

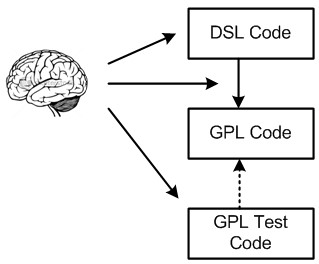
#### 4.3.2 Establishing the Correctness of the Execution Engine

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

In classical programming, we write the GPL code based on our understanding of the requirements. We then write unit tests, based on the same understanding, which test that code

(Fig. 4.4).

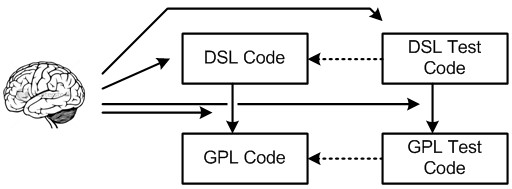


In DSL testing, we write the DSL, the DSL program and the execution engine based on our understanding of the requirements for the system. We can still write unit tests (in the GPL) based on this understanding to check for the correctness of the executing DSL program (Fig. 4.5).

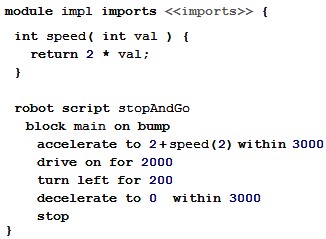
Writing one DSL program and one unit test ensures that this one program executes correctly regarding the test case. Our goal here is, however, to ensure that the transformation

|  |  |
| --- | --- |
| is correct *for all programs* we can write with the DSL. This can be achieved by writing many DSL programs and many tests – enough to make sure that every branch of the transformation is |  |
| covered at least once23. As always in testing, we encounter the |  |
| coverage problem: we have to write enough example programs and tests to cover all aspects of the language and the execution engine. In particular, we have to *first think of the corner cases* to write tests for them[[5]](#footnote-5). |  |

A variant of this approach is to express the test cases in the DSL (after extending the DSL with a way to express tests) and then executing the application code and the test code on the target platform together (Fig. 4.6). This is often more convenient, since the tests can be formulated more concisely on the level of the DSL. As we will see later, this approach is especially useful if we have *several* execution engines: we write the

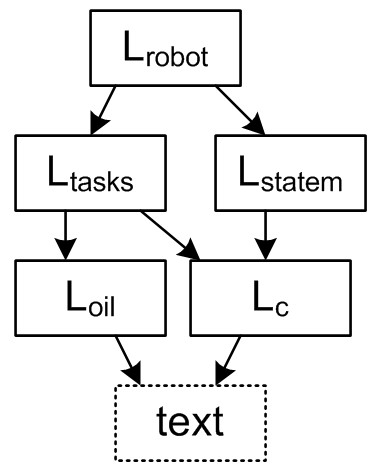
Figure 4.6: Test cases can also be expressed with the DSL and then executed on the target platform together with the application code.

test once and then execute it on all execution engines.

Note that the GPL program may have *additional, unintended* behaviors not prescribed by the DSL. These can often be exploited maliciously and are known as safety or security problems. These will not be found by testing the GPL code based on the requirements, but only by "trying to exploit" the program (known as *penetration testing*).

We will elaborate more on ensuring the correctness of the execution semantics in this chapter, as well as in the Part III chapter on DSL testing (Chapter 14), where we discuss the implementation aspects of DSL and IDE testing.

|  |  |
| --- | --- |
| *4.3.3 Transformation*  Transformations define the execution semantics of a DSL by mapping it to another language. In the vast majority of cases a transformation for *LD* recreates those patterns and idioms in *LD*−1 for which it provides linguistic abstraction. The result |  |

may be transformed further, until a level is reached for which a language with an execution infrastructure exists – often *D*0. Code generation, the case in which we generate GPL code from a DSL, is thus a special case in which *LD*0 code is generated.

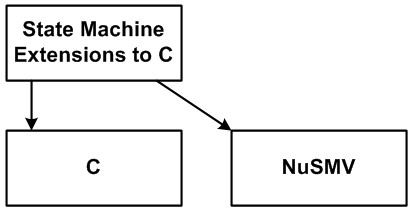
**mbeddr C:** The semantics of state machines are defined by their mapping back to C **switch** statements. This is repeated for higher *D* languages. The semantics of the robot control DSL (Fig. 4.7) is defined by its mapping to state machines and tasks (Fig. 4.8). To explain the semantics to

|  |  |
| --- | --- |
| the users, prose documentation is available as well. J |  |

**Component Architecture:** The component architecture DSL describes only structures: interfaces, components and systems. Many constraints about structural integrity are enforced, and a mapping to a distribution middleware is implemented. The formal definition of the semantics are implied by the mapping to the executable code. J

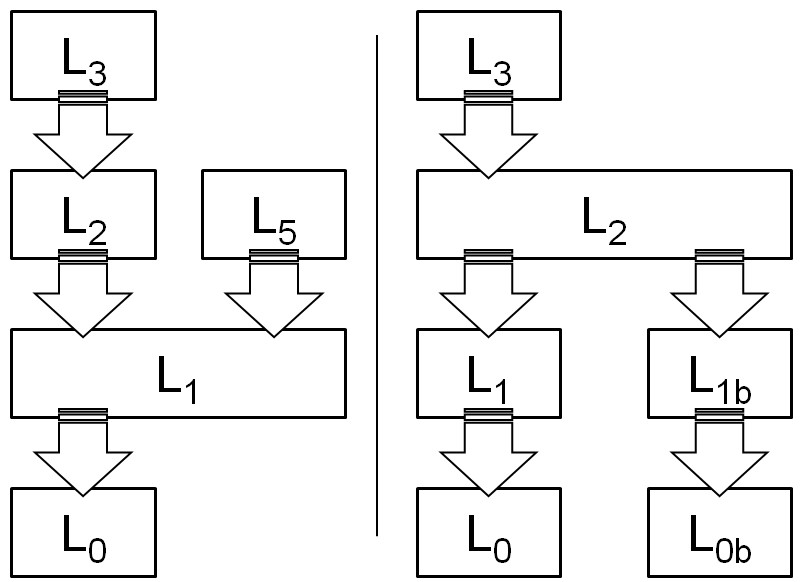
DSL programs may be mapped to *multiple* languages at the same time. Typically, there is one primary language that is used for execution of the DSL program (C in Fig. 4.9). The other languages may be used to configure the target platform (generated XML files) or provide input for verification tools (NuSMV in Fig. 4.9). In this case, one has to make sure that the semantics of all generated representations is actually the same. We discuss this problem in Section 4.3.7.

**mbeddr 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

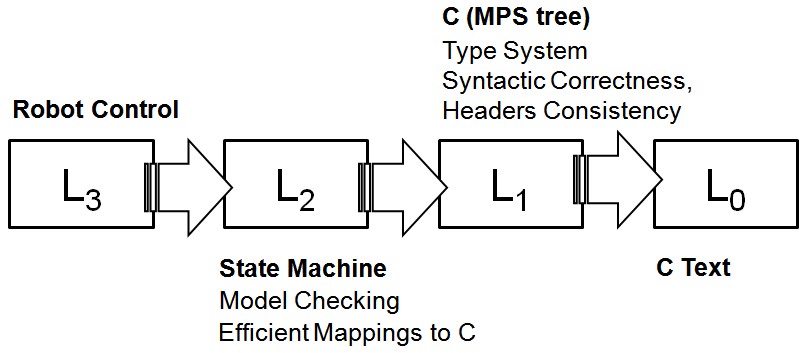


|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| from deadlocks, assuring liveness and specific safety properties such as "It will never happen that the out events **pedestrian light green** and **car light green** are set at the same time". J  *Multi-staged Transformation* There are several reasons why   |  |  | | --- | --- | | the gap between a language at *D* and its target platform may not be bridged by a single transformation. Instead, the overall transformation becomes a chain of subsequent transformations, an approach also known as *cascading*. |  | | Multi-staged transformation is a form of modularization, and so the reason for doing it is the same reason we always use for modularization: breaking down a big problem into a set of smaller problems that can be solved independently. In the case of transformations, this "big problem" is a big semantic gap between the DSL and the target language25. Modularization breaks down this big semantic gap into several smaller ones, making each of them easier to understand, test and main- |  | | tain26. |  | | Another reason for multi-stage transformations is the potential for reuse of each of the stages (Fig. 4.10). Reusing lower *D* languages and their subsequent transformations also im- |  | | plies reuse of potentially non-trivial analyses or optimizations that can be done at that particular abstraction level27. Con- |  | |

sider GPL compilers as an example. They can be retargetted relatively easily by exchanging the backends (machine code generation phases) or the frontend (programming language parsers and analyzers). For example, GCC can generate code for many different processor architectures (exchangeable backends), and it can generate backend code for several program-

ming 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. **mbeddr C:** The embedded C language (and some of its

|  |  |
| --- | --- |
|  | Figure 4.10: *Left:* Backend reuse. Dif- |
| higher *D* extensions) have various translation options, for several different target platforms (Win32 and Osek), an example of backend reuse. All of them are C code, but we generate different idioms in the code and different make files. J  Multi-stage transformation can also be a natural consequence of incremental language extension along the domain hierarchy, where we repeatedly build additional higher-level languages on top of lower-level languages. When transforming the higher-level languages, it is natural and obvious to transform them onto the next lower level, and not onto a language at *D*0.  **mbeddr C:** The extensions to C are all transformed back to C idioms during transformation. Higher-level DSLs, for example, a simple DSL for robot control, are reduced to C plus some extensions such as state machines and tasks  (Fig. 4.11), reusing the transformations for those abstractions back to C. J |  |

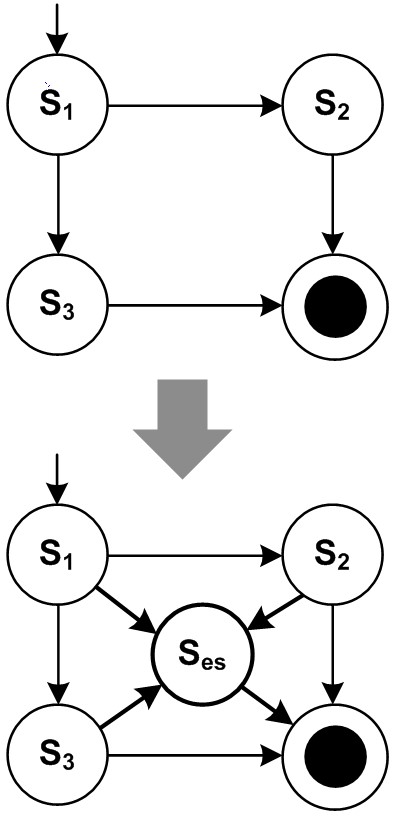


A special case of a multi-staged transformation is a preprocessor to a code generator. Here, a transformation reduces the set of used language concepts in a fragment to a minimal core, and only the minimal core is supported in the code generator. Note how, in this case, the source and target languages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

**mbeddr C:** Consider the case of a state machine to which 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. 4.12). Once done, the existing generator is run unchanged. You have effectively modularized the emergency stop concern into a preprocessor transformation. J

|  |  |
| --- | --- |
|  |  |
| 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. J  Multi-stage transformations can be challenging. It becomes harder to understand what is going on in total. Debugging the overall transformation can become hard, and good tool support |  |
| is needed28. |  |
| *Efficiency and Optimization* Transforming from *D* to *D*−1 allows the use of sophisticated optimizations, potentially resulting in very efficient code. DSL uses domain-specific abstractions and hence includes a lot of domain semantics, so optimizations can take advantage of this and produce very efficient *D*−1 code. However, building such optimizations can be very expensive. It is especially hard to build *global* optimizations that require knowledge about the structure or semantics of large or diverse parts of the overall program. Also, an optimization will always rely on some set of rules that determine when and how to optimize. There will always be corner cases where an experienced developer will be able to write more ef- |  |

**Component Architecture:** The DSL describes hierarchical



ficient *D*−1 code manually. However, this requires a competent developer and, usually, a lot of effort *for each specific program*. A tool (i.e. the transformation in this case) will typically address the 90% case well: it will produce reasonably efficient code in the vast majority of cases with very little effort (once the optimizations have been defined). In most cases, this is good enough – in the remaining corner cases, *D*−1 has to be written manually29.

|  |  |
| --- | --- |
| *Care about Generated Code* Ideally, generated code is a throwaway artifact, like object files in a C compiler. However, that’s not quite true. At least during development and test of the generator you may have to read, understand and debug the gener- |  |
| ated code. For incomplete DSLs30, i.e. those in which parts of the resulting program have to be written manually in *LD*−1, readability and good structure is even more important, because the manually written code has to be integrated with the generated parts of the *LD*−1 program. Hence, generated code should use meaningful abstractions, should be designed well, use good names for identifiers, be documented well, and be |  |
| indented correctly. In short, generated code should generally adhere to the same standards as manually written code. This also helps to diffuse some of the skepticism against code generation that is still widespread in some organizations. However, |  |
| there are several exceptions to this rule:  • Sometimes generating really well-structured code makes the generator *much* more complicated. You then have to decide whether you want to live with some less nicely structured generated code, or whether you want to increase generator complexity – a valid trade-off, since the generator also needs to be maintained! A good example is *import* statements when generating Java code. It can be a lot of work to find out exactly which imports are needed in a generated class. In this case it may be better to keep the generator simple and use fully qualified class names throughout the code, |  |
| and/or to import a few too many classes31. |  |

* Using a generator opens up additional options you wouldn’t consider when writing code manually (and which are hence considered ugly). An example is generated collection classes. Imagine that your models define entities, and from each entity you generate a Java Bean. In Java version 1.4 and earlier, Java did not have generics, so in order to work with collec-

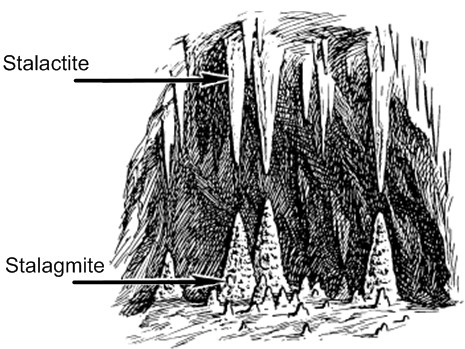
tions of entities you would use the generic **List** class. In the context of generated code you might want to consider generating a specific collection class for each entity, with an API typed to the respective Java Bean. This makes life much more convenient for those people who write Java code that *uses* the generated Beans.

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

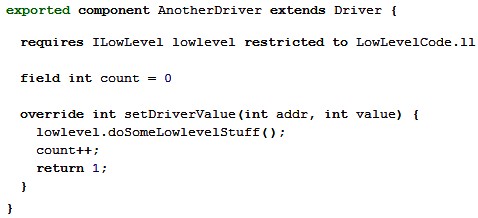
**mbeddr C:** The components extension to C supports components with provided and required ports. A required port declares which interface it is expected to be connected to. The same interface can be provided by different components, implementing the interface differently. Upon translation of the component extension, regular C functions are generated. An outgoing call on a required port has to be routed to the function that has been generated to implement the called interface operation in the target component. Since each component can be instantiated multiple times, and each instance can have its required ports connected to *different* component instances (implementing the same interface) there is no way for the generated code to know which particular function has to be called for an outgoing call on a required port for a given instance. An indirection through function pointers is used instead. Consequently, functions implementing operations in components take an additional **struct** as an argument, which provides those function pointers for each operation of each required port. A call on a required port is therefore a relatively ugly affair based on function pointers. However, to achieve the desired goal, no different, cleaner code approach is possible in C. It is optionally possible to *restrict* a required port to a particular component (Fig. 4.13). In this case, the target function is known statically and no function pointer-based indirection is required. The resulting code is cleaner and more efficient. Programmers trade flexibility for performance. J

The complexity can be reduced by splitting the overall transformation into several steps – see above. Another approach is to work with a manually implemented, rich domain specific platform. This 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 (or 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 in which you need to consider resource or timing constraints, or when the target platform is predetermined and perhaps limited, code generation is the better approach: trying to make the platform too generic or flexible will increase *its* complexity.



4

.

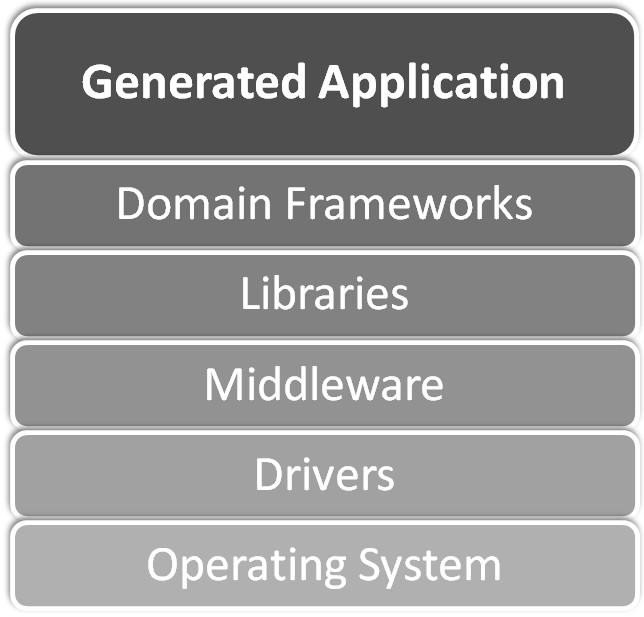
13

:

The required port

**ll**

*Platform*



**mbeddr C:** For most aspects, we use only a very shallow

platform. This is mostly for performance reasons and for the fact that the subset of C that is often used for embedded systems does not provide good means of abstraction. For example, state machines are translated to **switch** statements. If we were to generate Java code in an enterprise system, we might 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. J

#### 4.3.4 Interpretation

An interpreter is basically a program that acts on the DSL program it receives as an input. How it does that depends on the particular paradigm used (see Section 5.2). For imperative programs it steps through the statements and executes their side effects. In functional programs, the interpreter (recursively) evaluates functions. For declarative programs, some other evaluation strategy, for example based on a solver, may be used. We describe some of the details about how to design and implement interpreters in Section 12.

**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. J

**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. J

For interpretation, the domain hierarchy could be exploited as well: the interpreter for *LD* could be implemented in *LD*−1. However, in practice we see interpreters written in *LD*0. They may be extensible, so new interpreter code can be added to deal with the case where higher-level lanuguages add new language concepts.

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

#### 4.3.5 Transformation versus Interpretation

When defining the execution semantics for a language, a decision has to be made between transformation (code generation)

and interpretation. Here are some 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 *LD*−1. Interpreters are meta programs and as such harder to relate to existing code patterns.

|  |  |
| --- | --- |
|  |  |
| *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 generator33. Generally, performance is better in generated |  |
| environments, since interpreters always imply an additional layer of indirection during the execution of the program.  *Platform Conformance* Generated code can be tailored to any target platform. The code can look exactly as manually writ- |  |

*Debugging* Debugging generated code is straightforward if the code is well structured (which is up to the transformation) and an execution paradigm is used for which a decent debugging approach exists (not the case for many declarative approaches). Debugging interpreters is harder, because, they are meta programs. For example, setting breakpoints in the DSL program requires conditional breakpoints in the interpreter, which are typically cumbersome to use32

ten code would look; no support libraries are required. This is important for systems in which the source code (and not the DSL code) is the basis for a contractual obligations or for review and/or certification. Also, if artifacts need to be supplied to the platform that are not directly executable (descriptors, meta data), code generation is more suitable.

*Modularization* When incrementally building DSLs on top of existing languages, it is natural to use transformations to

*LD*.

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

*Runtime Change* In interpreted environments, the DSL program can be changed as the target system runs; the DSL editor can

even be integrated into the target system35.

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

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

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

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

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

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

#### 4.3.6 Sufficiency

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

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

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

#### 4.3.7 Synchronizing Multiple Mappings

|  |  |
| --- | --- |
| practice, this case often occurs if an interpreter is used in the IDE for "experimenting" with the models, and a code generator creates efficient code for execution in the target environment. To synchronize the semantics in this case, we recommend providing a set of test cases that are expressed on DSL level, and that are executed in all executable representations, expecting them to succeed in all of them. 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 se- |  |
| mantics are indeed the same37. |  |

Ensuring the semantics of the execution engine becomes more challenging if we transform the program to *several different* targets using several different transformations. We have to ensure that the semantics of all resulting programs are identical36. In

**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. J

**Refrigerators:** A similar situation occurs with the cooling DSL where an 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. J

#### 4.3.8 Choosing between Several Mappings

Sometimes there are several *alternative* ways in which a program in *LD* can be translated to a single *LD*−1, for example to realize different non-functional requirements (optimizations, target platform, tracing or logging). There are several ways in which 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 *LD* program and the annotation model as its input. There can be several different annotation models for the same core model that define several different transformations, to be used alternatively. An annotation model is a separate viewpoint (Section 4.4) an can hence be provided by a different stakeholder than the one who maintains the core *LD* program.
* Alternatively, *LD* can be extended to directly contain additional data to guide the decision. Since the data controlling the transformation is embedded in the core program, this is only useful if the DSL user can actually decide which alternative to choose, and if only one alternative should be chosen for each program. Annotation models provide more flexibility.
* Heuristics, based on patterns, idioms and statistics extracted from the *LD* program, can be used to determine the applicable transformation as well. Codifying these rules and heuristics can be hard though, so this approach is rarely used.

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

#### 4.3.9 Reduced Expressiveness and Verification

|  |  |
| --- | --- |
| It may be beneficial to limit the expressiveness of a language. Limited expressiveness often results in more sophisticated analyzability. For example, while state machines are not very |  |
| expressive (compared to fully fledged C), sophisticated formal verification algorithms are available (e.g., model checking using SPIN38 or NuSMV39). The same is true for first-order logic, where satisfiability (SAT) solvers40 can be used to check pro- |  |
| grams for consistency. If these kinds of analyses are useful for the model purpose, then limiting the expressiveness to the respective formalism may be a good idea, even if it makes expressing some programs in *D* more cumbersome41. Possibly a |  |
| DSL should be partitioned into several sub-DSLs, where some of them are verifiable and some are not. | . |

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

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

#### 4.3.10 Documentation

Formally, defining semantics happens by mapping the DSL concepts to *D*−1 concepts for which the semantics is known. For DSLs used by developers, and for domains that are defined inductively (bottom-up), this works well. For application domain DSLs, and for domains defined deductively (top-down), this approach is not necessarily good enough, since the *D*−1 concepts has no inherent meaning to the users and/or the domain. An additional way of defining the meaning of the DSL is required. Useful approaches include prose documentation42

as well as test cases or simulators. This way, domain users can "play" with the DSL and write down their expectations formally in test cases.

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

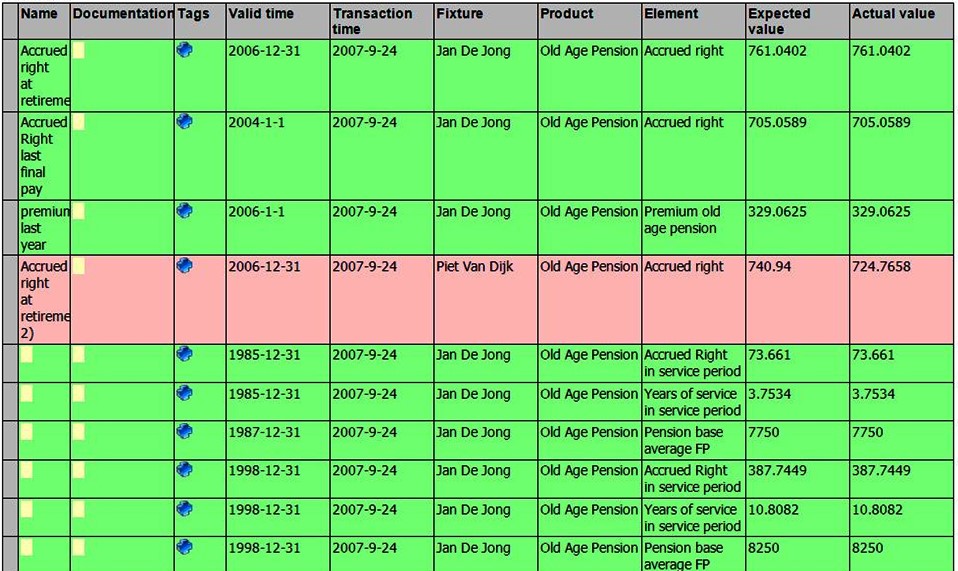
**Refrigerators:** This DSL has a separate viewpoint for defining test cases where domain experts can codify their expectations regarding the behavior of cooling programs. An interpreter is available to simulate the programs, observe their progress and stimulate them to see how they react. J

**Pension Plans:** This DSL supports an Excel-like tabular notation for expressing test cases for pension calculation rules (Fig. 4.16). The calculations are functional, and the calculation tree can be extended as a way of debugging the

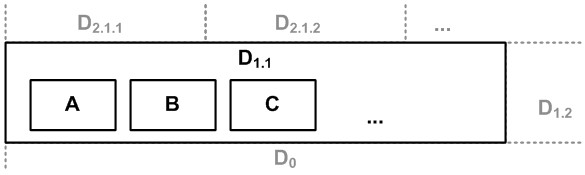
rules. J

### 4.4 Separation of Concerns

A domain *D* may be composed from different concerns. Each concern covers a different aspect of the overall domain. When developing a system in a domain, all the concerns in that domain have to be addressed. Separation into concerns is often driven by different aspects of the system being specified by



different stakeholders or at different times in the development process. Fig. 4.17 shows *D*1.1 composed from the concerns A, B and 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 concernspecific DSLs can be defined, each addressing one or more of the domain’s concerns43. 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 DSLs. Fig. 4.18 illustrates the two different approaches.

|  |
| --- |
| **mbeddr C:** The tasks language module includes the task |

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

**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 language44. Declarations in the languages can be combined in WebDSL modules. A WebDSL developer can choose how to factor declarations into modules; e.g., all access control rules in one module, or all aspects of some feature together in one module. J

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

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

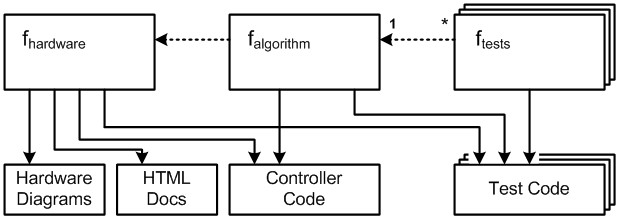
#### 4.4.1 Viewpoints for Concern Separation

If viewpoints are used, the concern-specific DSLs, and consequently the viewpoint models, should have well-defined dependencies; cycles should be avoided. If dependencies between

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

|  |  |
| --- | --- |
| 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 dependent viewpoint fragment (and the language to express it) have to provide a way of pointing to the referenced element. This usually means that the referenced element has | 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"). |
| to provide a qualified name that can be used in the reference45. | 45 In projectional editors one can *techni-* |
| 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 sep- | *cally* use the UUID of the target element for the reference, but for the user, some kind of qualified name is still necessary. |
| arate 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 the referenced fragment46. The viewpoint separation has to be aligned with the | 46 Projectional editors can use a different approach. They can store the information of all concerns in a single model, but then use different projections to address the needs of different stakeholders. This solves the problem of referential integrity. However, this |
| development process: the order of creation of the fragments must be aligned with the dependency structure. | approach does not support separate store and check in/out. |
| 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.  *4.4.2 Viewpoints as Annotation Models*  A special case of viewpoint separation is annotation models (already mentioned in Section 4.3.8). An annotation provides additional, often technical or transformation-controlling data | 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. |
| for elements in a core program47. This is especially useful in | 47 For those who know Eclipse EMF: |
| a multi-stage transformation (Section 4.3.3), 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 directly. Externalizing it into an annotation model solves that problem. |  |

**Refrigerators:** One concern in this DSL specifies the logical hardware structure of refrigerators installations. Another one describes the refrigerator cooling algorithm. Both



are implemented as separate viewpoints, where the algo-

rithm DSL references the hardware structure DSL. Using this dependency structure, different algorithms can be defined for the same hardware structure. Each of these algorithms resides in its own fragment. While the C code generation requires both behavior and hardware structure fragments, the hardware fragment is sufficient for a transformation that creates a visual representation of the hardware structures (see Fig. 4.19). J

**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 model48. J

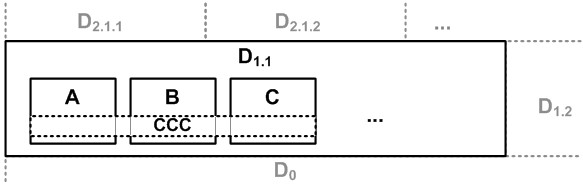
#### 4.4.3 Viewpoint Consistency

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

The other direction is more interesting. Assume two viewpoints: business data structure and persistence mapping. There may be a constraint that says that every **Entity** in the business data viewpoint has to have exactly one **EntityPersis-**

**tenceMapping** element that points to the respective **Entity**. It is an error if such an **EntityPersistenceMapping** does not exit. Checking this constraint has two problems:

|  |  |
| --- | --- |
| • The first problem may be performance. The *whole world* has |  |
| to be searched to check if a referencing program element exists somewhere. If the tool supports it, this problem can be solved by automatically maintained reverse indices49. |  |
| • The second problem is more fundamental: it is not clear what constitutes the whole world. The fragment with the persistence mapping for a given **Entity** may reside on a different machine or be under the control of a different user. It may not be accessible to the constraint checker when the user edits the business data fragment. To solve this problem, it is necessary to define explicitly what *the world* is, using some kind of configuration. For example, a C compiler’s include path or Java’s classpath are ways of defining the scope within which the overall system description must be complete. This does not necessarily have to be done by each developer who, for example, works on the business data. But at the point when the final system is generated or built, such a "world definition" is essential.  *4.4.4 Cross-Cutting Concerns*  In the discussion so far we have considered concerns that can be modularized clearly. Fig. 4.17 emphasizes this: the concern boxes are neatly arranged next to each other. However, there may also be concerns that do *not* fit into the chosen modularization approach. These are typically called *cross-cutting concerns*; see Fig. 4.20. |  |



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

*Handled by Execution Engine* If we are lucky, a concern that is cross-cutting in the domain can be handled completely by the execution engine. For example the collection of performance

data, billing information or audit logs typically does not have to be described in the DSL at all. Since every program in the domain has to address this concern in the same way, the implementation can be handled by the execution engine by inserting the respective code at the relevant locations (in the case of a generator).

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

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

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

*Cross-Cutting in the DSL* The third class is when the concern cross-cuts the programs written in the DSL and can *not* be modularized, as in the previous class. In this case we have to deal with cross-cutting concerns in the same way as we do today in programming languages: we either have to manually insert the code in *all* the relevant places in the DSL program, or we have to resort to aspect weaving on the DSL level50.

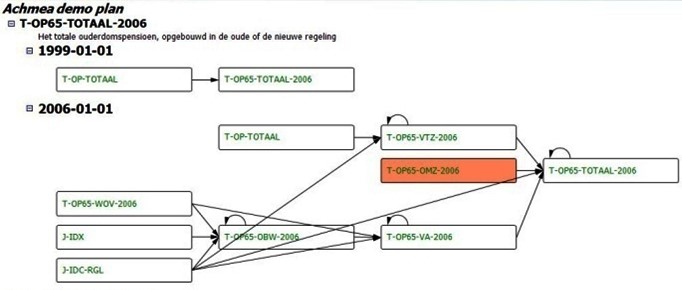
|  |
| --- |
| **Component Architecture:** We implemented a simple weaver that is able to introduce additional ports into existing components. It was used, among other things, to modularize the monitoring concern: if monitoring was enabled, this aspect component would add the **mon** port to all other components, enabling the **MonitoringConsole** to connect to the other components and query monitoring data (see the code below[[6]](#footnote-6)). J |

|  |
| --- |
| **namespace** monitoring **feature** monitoring {  **component** MonitoringConsole ...  **instance** monitor: ... **dynamic connect** monitor.devices .. .  **aspect** (\*) **component** { **provides** mon: IMonitoring  }  } |

#### 4.4.5 Views on Programs

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

**Pension Plans:** Pension plans can be shown in a graphical notation highlighting the dependency structure (Fig. 4.21). The dependencies can still be edited in this view, but the actual content of the pension plans is not shown. J



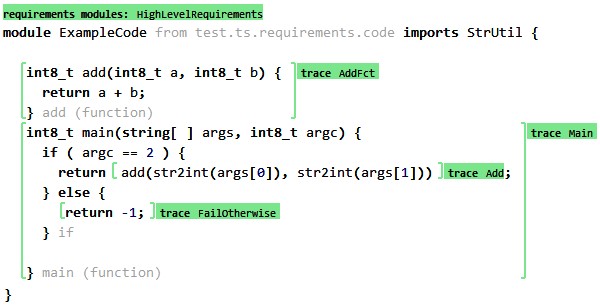
4

.

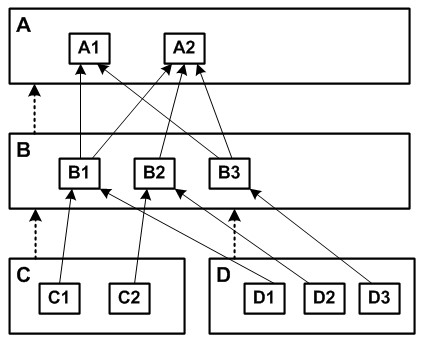
:

**mbeddr C:** Annotations are used for storing requirements .

traces and documentation in the models (Fig. 20.22). The program can be shown and edited with and without requirements traces and documentation text. J



|  |  |
| --- | --- |
| *4.4.6 Viewpoints for Progressive Refinement*  There is an additional use case for viewpoint models not related to the concerns of a domain, but to progressive refinement. Consider the development of complex systems, which typically proceeds in phases: it starts with requirements, proceeds to high-level component design and specification of nonfunctional properties, and finishes with the implementation of the components. In each of these phases, models can be used to represent the system with abstractions that are appropriate for the phase. An appropriate DSL is needed to represent the models in each phase (Fig. 4.23). The references between model el- |  |
| ements are called *traces*52. Since the same conceptual elements |  |
| may be represented on different refinement levels (e.g., component design and component implementation), synchronization between the viewpoint models is often required (see the next subsection). |  |



#### 4.4.7 Model Synchronization

|  |  |
| --- | --- |
| In some cases the rules for establishing consistency between viewpoints can be described formally, and hence the synchronization can be automated, if the DSL tool supports such syn- |  |
| chronization54. An example occurs in mbeddr C: |  |
| **mbeddr C:** Components implement interfaces. Each component provides an implementation for each method defined in each of the interfaces it implements. If a new method is added to an interface, all components that implement that particular interface must get a new, empty method implementation. This is an example of model synchronization. J  In this example the synchronization is trivial, for two reasons: first, there is a clear (unidirectional) dependency between the method implementation and the operation specification in the interface, so the synchronization is also unidirectional. Second, the information represented in both models/places is identical, so it is easy to detect an inconsistency and fix it. However, there are several more complicated cases:  • The dependency might be bidirectional, and changes may be allowed in either model. This means that two transformations have to be written, one for each direction, or a formalism for expressing the transformation has to be used that |  |
| can be executed in both directions55. In multi-user scenarios |  |

In the discussion of viewpoints so far we have assumed that there is no overlap between the viewpoints: every piece of information lives in exactly one viewpoint. Relationships between viewpoints are established by references (which means that the Referencing language composition technique can be used; this is discussed in Section 4.6.1). However, sometimes this is not the case, and the same (conceptual) information is represented in two viewpoint models. Obviously there is a constraint that enforces consistency between the viewpoints; the models have to be synchronized53.

it is also possible that the two models are changed at the same time, in an inconsistent way. In this case the changes have to be merged, or a clear priority (who will win) has to be established.

* The languages expressing the viewpoints may have been defined independent of each other, with *no dependency*. This probably means that it was discovered only *after the fact* that some parts of the model have to be synchronized. In this case the synchronization must be put into some kind of adapter language. It also means that the synchronization is

not as clean as if it had been "designed into" the languages (see the next item).

* In the mbeddr example, the information (the signature of the operation) was simply replicated, so the transformation was trivial. However, there may not be a 1:1 correspondence between the information in the two viewpoints. This makes the transformation more complex to write. In the worst case it may mean that the synchronization cannot be formally described and automated.

|  |  |
| --- | --- |
| opers have to express the correspondence (the trace links mentioned earlier) manually. However, consistency checks (and |  |
| possibly automatic synchronization) may still be possible, based on the manually expressed trace links.  In my work with DSLs I have only encountered the simplest cases of synchronization, which is why we don’t put much emphasis on this topic in the rest of the book. For more details, see the papers by Diskin57 and Stevens58.  *4.5 Completeness* |  |
| Completeness59 refers to the degree to which a language *L* can |  |
| express programs that contain all necessary information to execute them. An program expressed in an incomplete DSL requires additional specifications (such as configuration files or code written in a lower-level language) to make it executable.  Let us introduce a function *G* ("code generator") that transforms a program *p* in *LD* to a program *q* in *LD*−1. For a complete language, *p* and *q* have the same semantics, i.e. *OB*(*p*) == *OB*(*G*(*p*)) == *OB*(*q*) (see Section 4.3). For incomplete languages where *OB*(*G*(*p*)) ⊂ *OB*(*p*) we have to write additional |  |
| code in *LD*−1 to obtain a program in *D*−1 that has the same semantics as intended by the original program in *LD*. In cases in |  |

Sometimes the correspondence between models can only be expressed on an instance level (as in "This functional block corresponds to this software component")56. Consequently, devel-

which 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.

**mbeddr C:** The Embedded C language is complete regarding *D*−1, or even *D*−*m* for higher levels of *D*, since higher levels are always built as extensions of its *D*−1. Developers

can always fall back to *D*−1 to express what is not expressible directly with *LD*. Since the users of this system are developers, falling back to *D*−1 or even *D*0 is not a problem. J

#### 4.5.1 Compensating for Incompleteness

Integrating the *LD*−1 in the case of an incomplete *LD* language can be done in several ways:

• By calling "black box" code written in *LD*−1. This requires concepts in *LD* for calling *D*−1 foreign functions. No syntactic embedding of *D*−1 code is required, beyond the ability to call functions60.

|  |  |
| --- | --- |
| • By directly embedding *LD*−1 code in the *LD* program. This |  |
| is useful if *LD* is an extension of *LD*−1, or if the tool provides adequate support for embedding the *D*−1 language into *LD* programs. Note that *LD*−1 may not be analyzable, so mixing |  |
| *LD*−1 into *LD* code may compromise analyzability of the *LD* code.  • By using composition mechanisms of *LD*−1 to "plug in" the |  |
| manually written code into the generated code without actually modifying the generated files (also known as the Generation Gap61 pattern). 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 subclass62. 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.  • By inserting manually-written *LD*−1 code into the *LD*−1 code generated from the *LD* 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. |  |

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

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

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. Alterna-

tively, 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 4.1.2) which can be invoked as black boxes from DSL programs.

**WebDSL:** The core of a web application is concerned with persistent data and its presentation. However, web applications need to perform additional duties outside that core, for which useful libraries often 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. J

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 *LD*−1 to run in. That framework represents aspects of *D* outside the scope of *LD*.

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

*Controlling D*−1 *Code* Allowing users to manually write *D*−1 code, and especially if it is actually a GPL in *D*0, comes with two additional challenges. Consider the following example: the generator generates an abstract class from some model element. The developer is expected to subclass the generated

class and implement a couple of abstract methods. The manually written subclass needs to conform to a specific naming convention so that 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 name63?

|  |  |
| --- | --- |
| 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 Findbugs64 or |  |
| similar code checking tools. If one fails, an error message is reported to the developer. That error message can be worded by the developer of the DSL, helping the developer understand what exactly has to be done to solve the problem with the code.  *Semantic Consistency* As part of the definition of a DSL you will implement constraints that validate the DSL program in order to ensure some property of the resulting system (see Section 20.5). 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 one 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 |  |
| model65. |  |

**Component Architecture:** The Java code generator generates component implementation classes that use dependency injection to supply the targets for required ports. This ensures that the implementation class will have access to exactly those interfaces specified in the model. An alternative approach would be to simply hand to the implementation class some kind of factory or registry where a component implementation can look up instances of com-

ponents 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. J

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

#### 4.5.2 Roundtrip Transformation

Roundtrip transformation means that an *LD* program can be recovered from a program in *LD*−1 (written from scratch, or changed manually after generation from a previous iteration of the *LD* program). This is challenging, because it requires reconstituting the semantics of the *LD* program from idioms or patterns used in the *LD*−1 code. This is the general reverse engineering problem and is not generally possible, although progress has been made over recent years (see for example66).

Note that for complete languages roundtripping is generally not useful, because the complete program can be written in *LD* 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 *LD*−1 program, then the generated *LD*−1 program is so complicated that manually changing the *D*−1 code in a consistent and correct way is tedious and error-prone.

Roundtripping has traditionally been used with respect to UML models and generated class skeletons. In that case, the abstractions between the model and the code are similar (both are classes); the tool basically just provides a different concrete syntax (diagrams). This similarity of abstractions in the code

|  |
| --- |
| and the model made roundtripping possible to some extent. However, it also made the models relatively useless, because they did *not* provide a significant benefit in terms of abstraction over code details. We generally recommend avoiding any attempt to build support for roundtripping.  **mbeddr 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. J |

**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 proved to provide good coverage, the need to "tweak" the generated code has not come up. J

**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. J

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

### 4.6 Language Modularity

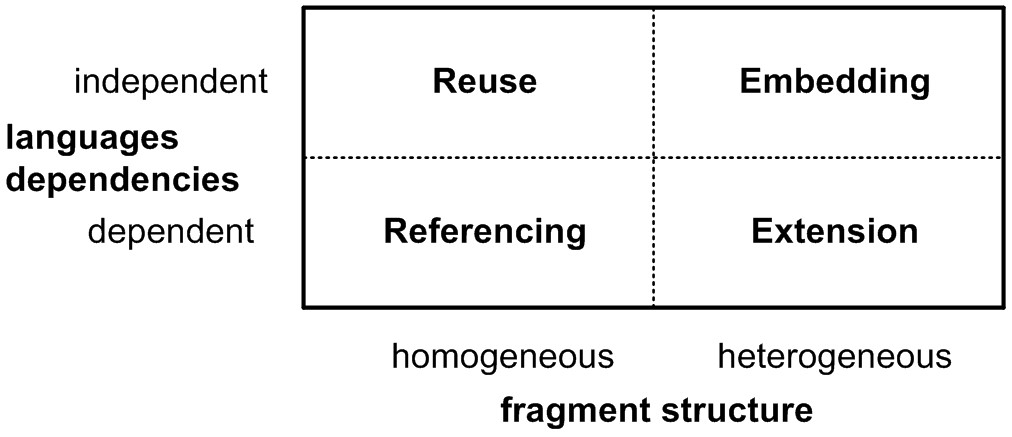
Reuse of modularized parts makes software development more efficient, since similar functionality does not have to be developed over and over again. A similar argument can be made for languages. Being able to reuse languages, or parts of languages, in new contexts makes designing DSLs more efficient.

|  |  |
| --- | --- |
| Language composition requires the composition of abstract |  |
| syntax, concrete syntax, constraints/type systems and the execution semantics67. We discuss all of these aspect in this |  |
| section. However, in the discussion of semantic integration, we consider only the case in which the composed language |  |
| uses the same (or closely related) behavioral paradigms68, since |  |

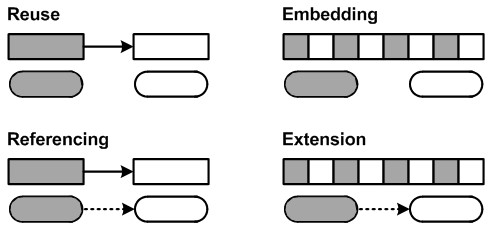
otherwise the composition can become very challenging. We mostly focus on imperative programs. We discuss behavioral paradigms in more detail in Section 5.

*Composition Techniques* We have identified the following four composition strategies: referencing, extension, reuse and embedding. We distinguish them regarding fragment structure

and language dependencies, as illustrated in Fig. 4.24. Fig. 4.25 shows the relationships between fragments and languages in these cases69.

.

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



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

*More Detailed Examples* Part III of the book discusses the implementation of these modularization techniques with various tools (Section 16). As part of this discussion we present much more concrete and detailed examples of the various composition techniques. You may want to take a look at those examples while you read this section.

#### 4.6.1 Language Referencing

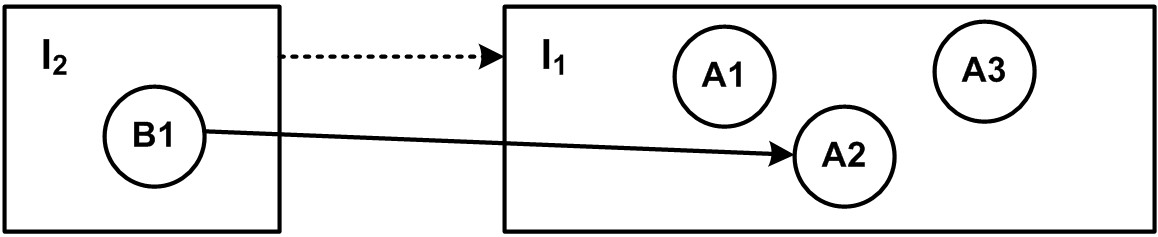
Language referencing enables *homogeneous* fragments with crossreferences among them, using *dependent* languages (Fig. 4.26).

A fragment *f*2 depends on *f*1. *f*2 and *f*1 are expressed with different languages *l*2 and *l*1. The referencing language *l*2 depends on the referenced language *l*1 because at least one concept in the *l*2 references a concept from *l*1. We call *l*2 the *referencing* language, and *l*1 the *referenced* language. While equations

(1.2) and (1.3) (see Section 3.3) continue to hold, (1.1) does not. Instead:

∀*r* ∈ *Refsl2* | *lo*(*r*.*from*) = *l*2 ∧ (*lo*(*r*.*to*) = *l*1 ∨ *lo*(*r*.*to*) = *l*2)

(4.1)



*Viewpoints* As we have discussed before in Section 4.4, a domain *D* can be composed from different concerns. One way of dealing with this is to define separate concern-specific DSLs, each addressing one or more of the domain’s concerns. A program then consists of a set of concern-specific fragments, which relate to each other in a well-defined way using language referencing. This approach has the advantage that different stakeholders can modify "their" concern independent of others. It also allows reuse of the independent fragments and languages with different referencing languages. The obvious drawback is that for tightly integrated concerns the separation into separate fragments can be a usability problem.

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

**Refrigerators:** As an example, consider the domain of refrigerator configuration. The domain consists of three concerns. The first concern *H* describes the hardware structure of refrigerator appliances including compartments, compressors, fans, valves and thermometers. The second concern *A* describes the cooling algorithm using a statebased, asynchronous language. Cooling programs refer to hardware building blocks and access 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* → *H* and *T* → *A*. Each of these concerns is 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. J

*Progressive Refinement* Progressive refinement, also introduced earlier (Section 4.4.6), also makes use of language referencing.

#### 4.6.2 Language Extension

Language extension enables *heterogeneous* fragments with *dependent* languages (Fig. 4.27). A language *l*2 extending *l*1 adds additional language concepts to those of *l*1. We call *l*2 the *extending* language (or language extension), and *l*1 the *base* language. To allow the new concepts to be used in the context provided by *l*1, some of them extend concepts in *l*1. So, while

*l*1 remains independent, *l*2 becomes dependent on *l*1, since:

∃*i* ∈ *Inh*(*l2*) | *i*.*sub* = *l*2 ∧ *i*.*super* = *l*1 (4.2)

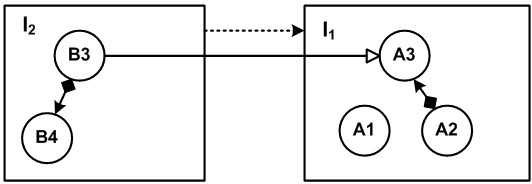
Consequently, a fragment *f* contains language concepts from both *l*1 and *l*2:

∀*e* ∈ *Ef* | *lo*(*e*) = *l*1 ∨ *lo*(*e*) = *l*2 (4.3)

In other words, *Cf* ⊂ (*Cl*1 ∪ *Cl*2), so *f* is *heterogeneous*. For heterogeneous fragments (1.3) does not hold anymore, since:

∀*c* ∈ *Cdnf* | (*lo*(*co*(*c*.*parent*)) = *l*1 ∨ *lo*(*co*(*c*.*parent*)) = *l*2)∧

(*lo*(*co*(*c*.*child*)) = *l*1 ∨ *lo*(*co*(*c*.*child*)) = *l*2) (4.4)



Language extension fits well with the hierarchical domains introduced in Section 3.1: a language *LB* for a domain *D* may extend a language *LA* for *D*−1. *LB* 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 *LA* for *D*. Language semantics are typically defined by mapping the new abstractions to just these idioms (see Section 4.3) *inline*. This process, also known as *assimilation*, transforms a heterogeneous fragment (expressed in *LD* and *LD*+1) into a homogeneous fragment expressed only with *LD*.

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.

**mbeddr C:** As an example, consider embedded programming. The C programming language is typically used as the GPL for *D*0 in this case. Extensions for embedded programming include state machines, tasks or data types with physical units. Language extensions for the subdomain of real-time systems may include ways of specifying deterministic scheduling and worst-case execution time. For the avionics subdomain support for remote communication using some of the bus systems used in avionics could be added. J

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

* *Extension Flavor* In the first case we provide (a little, local) additional syntax to an otherwise unchanged language. For example, C may be extended with new data types and literals for complex numbers, as in **complex c = (3+2i);**. The programs still essentially look like C programs, with specific extensions in a few places.
* *Embedding Flavor* The other case is where we create a completely new language, but reuse some of the syntax provided by the base language. For example, we could create a state machine language that reuses C’s expression and types in guard conditions. This use case *feels* like embedding (we embed syntax from the base language in our new language), but in the classification according to syntactic integration and dependencies, it is still extension. Embedding would prevent dependencies between the state machine language and C.

Language extension is also a very useful way to address the problem of DSLs often starting as simple, but then becoming more complicated over time, because new corners or intricacies in the domain are discovered as users gain more experience in the domain. These corner cases and intricacies can be factored into a separate language module that extends the core DSL. The use of these extensions can then initially be restricted to a few users in order to find out if they are really needed. Different experiments can even be performed at the same time,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| with different groups of users using different extensions. Even once these extensions have proved useful, "advanced" language features can be restricted in this way to a small group of "advanced" users who handle the hard cases by using the extension.  Incremental extension can help to avoid the feared "customization cliff". The customization cliff is a term introduced   |  |  | | --- | --- | | by Steve Cook : *once you step outside of what is covered by your* |  | | *DSL, you plunge down a cliff onto the rocks of the low-level plat-* |  |   *form.* If DSLs are built as incremental extensions of the next |

lower language, then stepping outside any DSL on level *D* will only plunge you down to the language for *D*−1. And presumably you can always create an additional extension that extends your DSL to cover an additional, initially unexpected aspect.

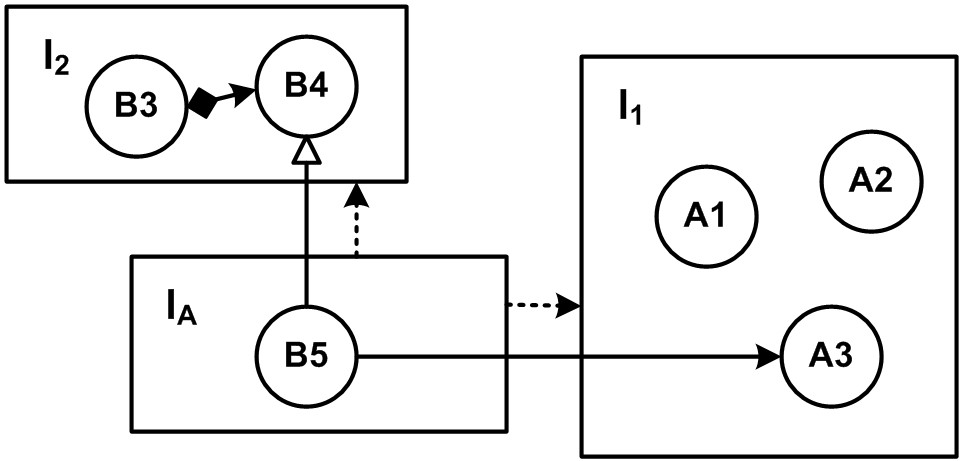
Defining a *D* language as an extension of a *D*−1 language can also have drawbacks. The language is tightly bound to the *D*−1 language it is extended from. While it is possible for a stand-alone DSL in *D* to generate implementations for different *D*−1 languages, this is not easily possible for DSLs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

*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, or 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).

**mbeddr C:** Modules can be marked as *MISRA-compliant*, which prevents the use of those C constructs that are not

### 71

|  |  |
| --- | --- |
| allowed in MISRA-C . Prohibited concepts are reported |  |
| as errors directly in the program. J  *4.6.3 Language Reuse*  Language reuse enables *homogenous* fragments with *independent* languages (Fig. 4.28). 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:  ∃*r* ∈ *Refsf2* | *fo*(*r*.*from*) = *f*2 ∧ |  |
| (*fo*(*r*.*to*) = *f*1 ∨ *fo*(*r*.*to*) = *f*2) (4.5)  Since *l*2 is independent, it cannot directly reference concepts |  |
| in *l*1. This makes *l*2 reusable with different languages (in contrast to language referencing, where concepts in *l*2 reference concepts in *l*1). We call *l*2 the *context* language and *l*1 the *reused* language.  One way of realizing dependent fragments while retaining independent languages is using an adapter language *lA* where *lA extends l*2, and:  ∃*r* ∈ *RefslA* | *lo*(*r*.*from*) = *lA* ∧ *lo*(*r*.*to*) = *l*1 (4.6) |  |



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

**Example:** Consider a language for describing user interfaces. It provides language concepts for various widgets, layout definition and disable/enable strategies. It also supports data binding, where data structures are associated with widgets, to enable two-way synchronization between the UI and the data. Using language reuse, the same UI language can be used with *different* data description languages. Referencing would not achieve this goal, because the UI language would have a direct dependency on a particular data description language. Changing the dependency direction to *data* → *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). J

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. J

#### 4.6.4 Language Embedding

Language embedding (Fig. 4.29) 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:

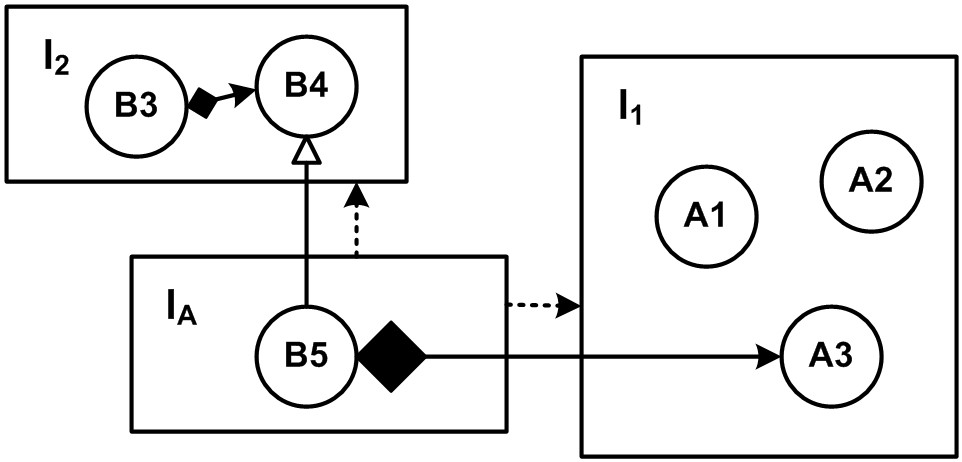
∀*c* ∈ *Cdnf* | *lo*(*co*(*c*.*parent*)) = *l*1 ∧

(*lo*(*co*(*c*.*child*)) = *l*1 ∨ *lo*(*co*(*c*.*child*)) = *l*2)) (4.7)

Unlike language extension, where *l*2 depends on *l*1 because concepts in *l*2 extends concepts in *l*1, there is no such dependency in this case. Both languages are independent. We call *l*2 the *embedded* language and *l*1 the *host* language. Again, an adapter language *lA* that extends *l*1 can be used to achieve this, where:

∃*c* ∈ *CdnlA* | *lo*(*c*.*parent*) = *lA* ∧ *lo*(*c*.*child*) = *l*1 (4.8)

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



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

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

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

Just as in the embedding-flavored extension case (cf. Section 4.6.2), sometimes the embedded language must also be restricted. If you embed the C expression language in state machine guard conditions, you may want to restrict the user from using pointer types or all the expressions related to pointers in C.

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

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

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

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

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

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

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

**mbeddr 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 the part on MPS in Section 16.2.7. As a consequence of how MPS attributes work, these meta data can be applied to program elements defined in any arbitrary language. J

#### 4.6.5 Implementation Challenges and Solutions

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

*Syntax* Referencing and Reuse keeps fragments homogeneous. Mixing of concrete syntax is not required. A reference between fragments is usually simply an identifier and does not have its own internal structure for which a grammar would be required72. The name resolution phase can then create the

actual cross-reference between abstract syntax objects.

|  |
| --- |
| simple, dotted names such as **compartment1.valve**. J  **Example:** In the UI example, the adapter language simply introduces dotted names to refer to fields of data structures. J  Extension and embedding requires modular concrete syntax definitions because additional language elements must be mixed with programs written with the base/host language. As we discuss in Part III (mostly in Section 7), combining independently developed languages after the fact can be a problem: depending on the parser technology, the combined grammar may not be parsable with the parser technology at hand. There are parser technologies that do not exhibit this problem, and projectional editors avoid it by definition. However, several widely used language workbenches have problems in this respect.  **mbeddr C:** State machines are hosted in regular C programs. This works because the C language’s **Module** con- |

**Refrigerators:** The algorithm language contains cross-references into the hardware language. Those references are ified names, in which case the strings use dots and colons. However, this is still a trivial token structure, so it is acceptable to define the structure separately in both languages.

struct contains a collection of **IModuleContents**, and the **StateMachine** concept implements the **IModuleContent** concept interface. This state machine language is designed specifically to be embedded into C, so it can access and extend **IModuleContent** (Fig. 4.30). If the state machine language were embeddable in any host language in addition to C, this dependency on **ModuleContent** (from the C base language) would not be allowed. An adapter language would have to be created which adapts a **StateMachine** to **IModuleContent**. J

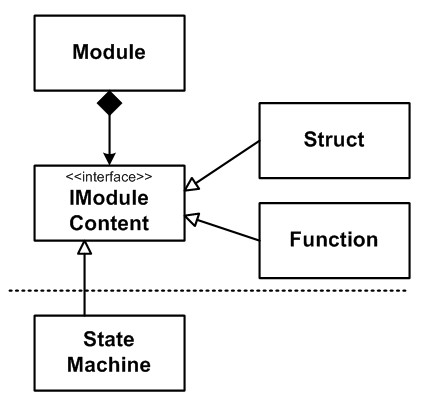
*Type Systems* For referencing, the type system rules and constraints of the referencing language typically have to take into account the referenced language. Since the referenced language is known when developing the referencing language, the type system can be implemented with the referenced language 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 whether it is on or off). J

In the 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.

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

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



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

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

Three cases have to be considered for referencing. The first one (Fig. 4.31) propagates the referencing structure to the target fragments. We call these two transformations *single-sourced*, since each of them only uses a single, homogeneous fragment as input and creates a single, homogeneous fragment as output, typically with references between them. Since the referencing language is created with knowledge about the referenced language, the generator for the referencing language can be written with knowledge about the names of the elements that have to be referenced in the fragment generated from the referenced fragment. If a generator for the referenced language already exists, it can be reused unchanged. The two generators basically share knowledge about the names of generated elements.

**Component Architecture:** In the types viewpoint, interfaces and components are defined. The types viewpoint is independent, and it is sufficient for the generation of the code necessary for implementing component behavior: Java base classes are generated that act as the component implementations (expected to be extended by manually written subclasses). A second, dependent viewpoint describes component instances and their connections; it depends on the types viewpoint. A third 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

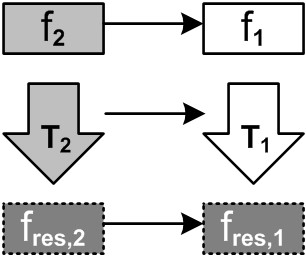
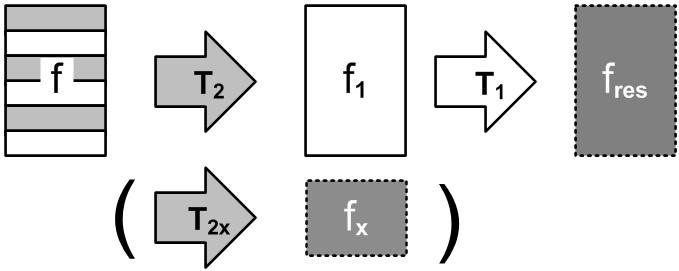
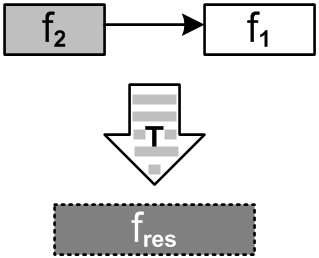


Figure 4.31: Referencing: Two separate, dependent, single-source transformations

those generated (and hand-written) classes. J

The second case (Fig. 4.32) is a multi-sourced transformation that creates one single homogeneous fragment. This typically occurs if the referencing fragment is used to guide the transformation of the referenced fragment, for example by specifying transformation strategies (annotation models). In this case, a new transformation has to be written that takes the referencing fragment into account. The possibly existing generator for the referenced language cannot be reused as is.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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. J  The third case, an alternative to rewriting the generator, is the use of a preprocessing transformation (Fig. 4.33), that changes the referenced fragment in a way consistent with what the referencing fragment prescribes. The existing transformations for the referenced fragment can then be reused.   |  |  | | --- | --- | |  |  | | 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 *LD* to *LD*−1, a pro- |  |   cess also called assimilation. In other words, transformations for language extensions map a heterogeneous fragment (containing *LD*−1 and *LD* code) to a homogeneous fragment that contains only *LD*−1 code (Fig. 4.34). In some cases additional |

files may be generated, often configuration files. In any case, the subsequent transformations for *LD*−1, if any, can be reused unchanged.

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

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

tension language. Optionally, additional files are generated, often configuration files.

by transformations defined in another language.

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

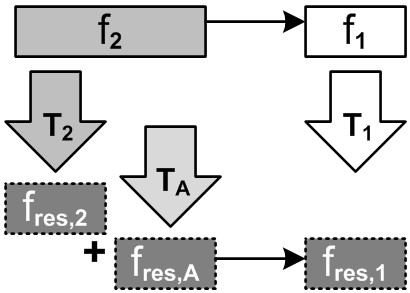
Language extension introduces the risk of semantic interactions. The transformations associated with several independently developed extensions of the same base language may interact with each other. To avoid the problem, transformations should be built in a way so that they do not "consume scarce resources" such as inheritance links73.

**Example:** Consider the (somewhat artificial) 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. J

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

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. 4.35, describes the case in which 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 context language.



For example, the generator of the context language may generate a class with abstract methods; the adapter may generate a subclass and implement these abstract methods.

In the second case, Fig. 4.36, the existing generator for the reused fragment has to be enhanced with transformation code specific to the context language. A mechanism for composing transformations is needed.

The third case, Fig. 4.37, 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.

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

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. 4.38 illustrates this case.

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

#### 4.7 Concrete Syntax

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

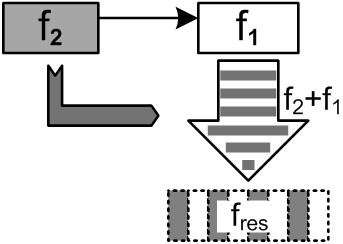
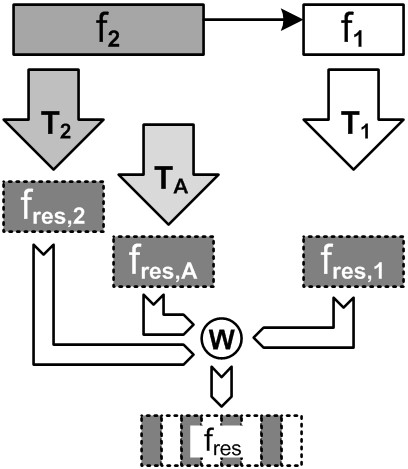
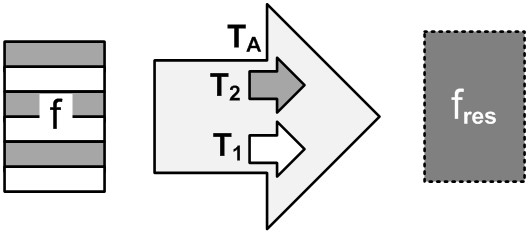


Figure 4.36: Reuse: composing transformations





##### 4.7.1 Design Concerns for Concrete Syntax

|  |  |
| --- | --- |
|  |  |
| *Writability* A writable syntax is one that can be written efficiently. This usually means that the syntax is concise, because users have to type less. However, a related aspect is tool support: the degree to which the IDE can provide bet- |  |
| ter editing support75 (code completion and quick fixes in |  |
| particular) makes a difference to readability.  *Readability* A readable syntax means that it can be read effectively. A more concise syntax is not necessarily more read- |  |
| able, because context may be missing76, in particular for |  |
| people other than those who have written the code.  *Learnability* A learnable syntax is useful to novices in particular, because it can be "explored", often exploiting IDE sup- |  |
| port77. For example, the more the language uses concepts |  |
| that have a direct meaning in the domain, the easier it is for domain users to lean the language.  *Effectiveness* Effectiveness relates to the degree that a language enables routine users to effectively express typical domain problems *after* they have learned the language.  *Tradeoffs* It is obvious that some of these concerns are in conflict. A very writable language may not be very readable. If a group of stakeholders **R** uses artifacts developed by another group **W** (e.g. by referencing some of the program elements), it is important that a readable language is used. A learnable language may feel "annoyingly verbose and cumbersome" to |  |
| routine users after a while78. However, creating an effective |  |

In particular the following concerns may be addressed when designing a concrete syntax74:

syntax and trying to convince users to adopt the language even though it is hard(er) to learn may be a challenge.

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

*Multiple Notations* One way to solve these dilemmas is to provide different concrete syntaxes for the same abstract syntax, and let users choose. For example, beginners can chose a more learnable one, and switch to a more effective one over

time. However, depending on the tooling used, this can be a lot of work.

*Multiple Notations* For projectional editors it is relatively easy to define several notations for the same language concept. By changing the projection rules, existing programs can be shown in a different way. In addition, different notations (possibly showing different concerns of the overall program)

can be used for different stakeholders.

**mbeddr 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. A graphical notation will be added in the future, as MPS’ support for graphical notations improves. J

**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. J

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

|  |  |
| --- | --- |
| *Reports and Visualization* A visualization is a graphical representation of a model that cannot be edited. It is created from the core model using some kind of transformation, and highlights a particular aspect of the source program. It is often automatically laid out79. The resulting diagram may be static |  |
| (i.e. an image file is generated) or interactive (where users can show, hide and focus on different parts of the diagram). It may provide drill-down back to the core program (double-clicking on the figure in the image opens the code editor at the respective location)[[7]](#footnote-7). |  |

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

Visualizations and reports are a good way of resolving a potential conflict if the primary DSL users want to use a writable notation and other stakeholders want a more readable representation. Since reports and visualizations are not the primary

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *4.7.2 Classes of Concrete Syntax*   |  |  | | --- | --- | | There are a couple of major classes for DSL concrete syntax81: |  | | *textual* DSLs use linear textual notations, typically based on ASCII or Unicode characters. They basically look and feel like |  | |

traditional programming languages. *Graphical* DSLs use graphical shapes. An important subgroup is represented by those that use box-and-line diagrams that look and feel like UML class diagrams or state machines. However, there are more options for graphical notations, such as those illustrated by UML timing diagrams or sequence diagrams. *Symbolic* DSLs are textual DSLs with an extended set of symbols, such as fraction bars, mathematical symbols or subscript and superscript. *Tables and matrices* are a powerful way to represent certain kinds of data and can play an important part for DSLs.

The perfect DSL tool should support free combination and integration of 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.

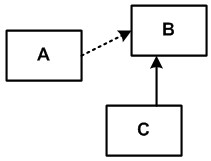
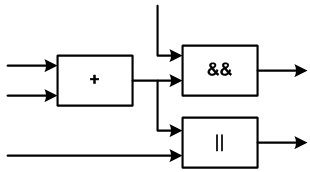


Figure 4.40: Graphical notation for relationships.



*When to Use Which Form* We do not want to make this section a complete discussion between graphical and textual DSLs – a discussion that is often heavily biased by previous experience, prejudice and tool capabilities. Here are some rules of thumb. Purely textual DSLs integrate well with existing development infrastructures, making their adoption relatively easy. They are well suited for detailed descriptions, anything that is algorithmic or generally resembles (traditional) program source code. A good textual syntax can be very effective (in terms of the design concerns discussed above). Symbolic notations can be considered "better textual", and lend themselves to domains that make heavy use of symbols and special notations; many scientific and mathematical domains come to mind. Tables are very useful for collections of similarly structured data items, or for expressing how two independent dimensions of data relate. Tables emphasize readability over writability. Finally, graphical notations are very good for describing relationships (Fig. 4.40), flow (Fig. 4.41) or timing and causal relationships (Fig. 4.42). They are often considered easier to learn, but may be perceived as less effective by experienced users.

Figure 4.41: Graphical notation for flow

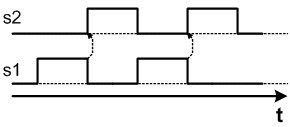
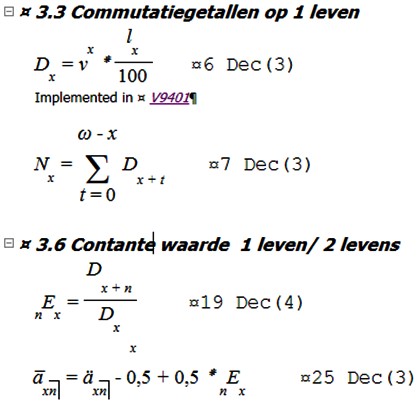


Figure 4.42: Graphical notation for causality and timing



**Pension Plans:** The pension DSL uses a mathematical notation to express insurance mathematics (Fig. 4.43). 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. J

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **mbeddr C:** The core DSLs use mostly textual notations with some tabular enhancements, for example for decision tables (Fig. 20.4). However, as MPS’ capability for handling graphical notations improves, we will represent state machines as diagrams. J   |  |  | | --- | --- | | Selection of a concrete syntax is simple for domain user DSLs if there is an established notation in the domain. The challenge then is to replicate this notation as closely as possible with the DSL, while cleaning up possible inconsistencies in the notation  (since presumably it had not been used formally before). I like |  | | to use the term "strongly typed (Microsoft) Word" in this case82. |  | |

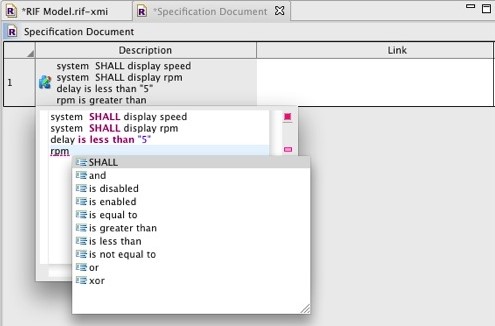
For DSLs targeted at developers, a textual notation is usually a good starting point, since developers are used to working with text, and they are very productive with it. Tree views and some visualizations are often useful for outlines, hierarchies or overviews, but not necessarily for editing. Textual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

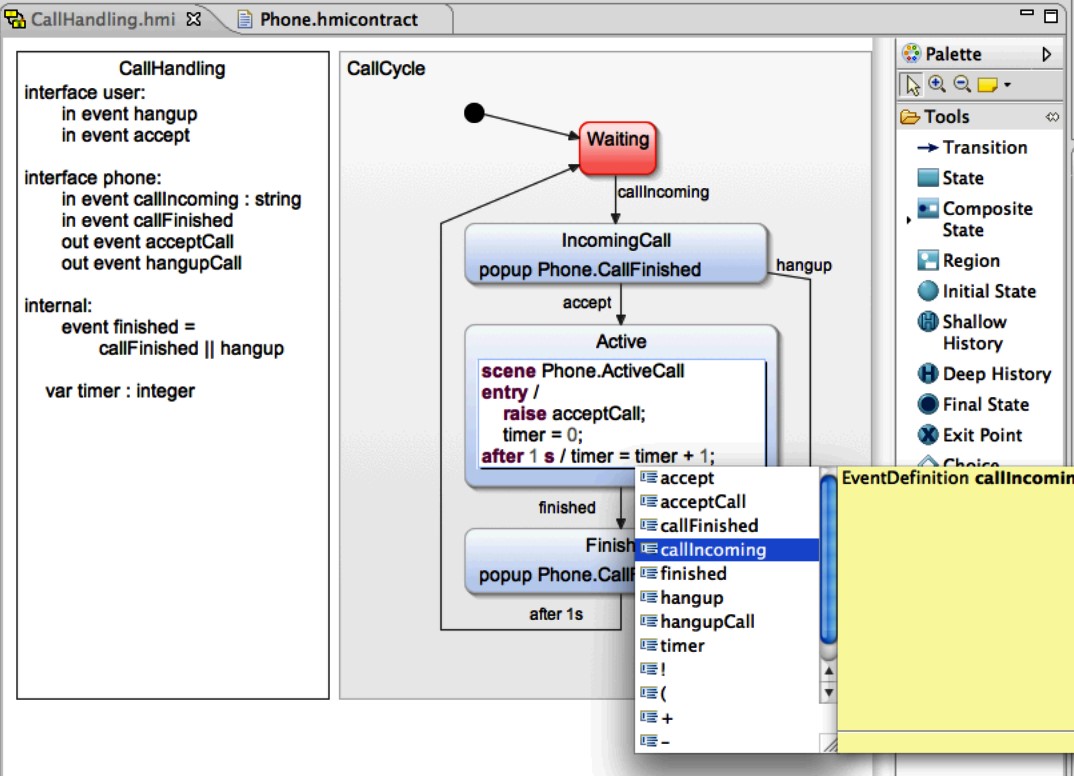
**mbeddr C:** C is the baseline for embedded systems, and everybody is familiar with it. A textual notation is useful for many concerns in embedded systems. Note that several languages create visualizations on the fly, for example for module dependencies, component dependencies and component instance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

There are very few DSLs where a *purely* graphical notation makes sense, because in most cases some textual languages are embedded in the diagrams or tables: state machines embedded expressions in guards and statements in actions (Fig. 20.7); component diagrams use text for specifications of operations in interfaces, maybe using expressions for preconditions; block diagrams use a textual syntax for the implementation/parametrization of the blocks (Fig. 4.45); tables may embed textual

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | notations in the cells (Fig. 4.46). Integrating textual languages |  | | into graphical ones is becoming more and more important, and tool support is improving. |  |   be used.  Figure |

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

is n

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

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

When entering this query the IDE cannot provide code completion for the fields after the **SELECT** because at this point the table has not yet been specified. A more suitable syntax, with respect to IDE support, would be:

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

It is better because now the IDE can provide support code completion for the fields based on the table name that has already been entered when you specify the fields83.

Another nice example is dot-notation for function calls. Consider a functional language. Typical function call syntax is **f(a, b, c)** or possiby **(f a b c)**. In either case, the function comes first. Now consider a notation where you can (optionally) write the first argument before the dot, i.e. **a.f(b, c)**. This has a significant advantage in terms of IDE support: after the user enters **a.**, code completion can propose all the functions that are available for the type of **a**. This leads to much better explorability of the language compared to the normal functionfirst syntax: since at the time of writing the function, the user has not yet written the value on which to apply the function, the IDE can provide no support[[8]](#footnote-8).

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

*Relationship to Hierarchical Domains* Domains at low *D* are most likely best expressed with a textual or symbolic concrete syntax. Obvious examples include programming languages at *D*0. Mathematical expressions, which are also very dense and algorithmic, use a symbolic notation. As we progress to higher *D*s, the concepts become more and more abstract, and as state machines and block diagrams illustrate, graphical notations become useful. However, these two notations are also a good example of language embedding, since both of them require expressions: state machines in guards and actions (Fig. 20.7), and block diagrams as the implementation of blocks (Fig. 4.45 and Fig. 5.5). Reusable expression languages should be embedded into the graphical notations. If 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 use a textual notation to provide "implementation details" for the structures defined by the graphical viewpoint[[9]](#footnote-9).

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

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. - [↑](#footnote-ref-3)
4. y [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. . [↑](#footnote-ref-6)
7. . [↑](#footnote-ref-7)
8. . [↑](#footnote-ref-8)
9. . [↑](#footnote-ref-9)