finitive ANTLR Reference: Building Domain-Specifi
c Languages. Pragmatic Programmers, May 2007.

MARKUS VOELTER, VOELTER@ACM.ORG

DOMAIN-SPECIFIC LANGUAGE DESIGN

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

0.1 Introduction

Efficiently bridging the gap between domain concepts and the implementation of these concepts in a programming language is one of the core challenges of software engineering. Modern high-level programming languages considerably reduce this gap, compared to the assembly programming of yore. However, as Alan J. Perlis stated in one of his famous epigrams on programming, "a programming language is low level if its programs require attention to the irrelevant" [1]. For many tasks, high-level programming languages still require attention to the irrelevant. Since Turing we know that we need only a minimalistic language to express any computation. Based on this principle, general purpose programming languages provide generic mechanisms such as procedural abstraction that can be used for a wide range of programs. However, this requires the programmer to encode knowledge about a particular domain using these generic mechanisms. The programming environment — compiler and integrated development environment (IDE) — has no particular knowledge about the domain and cannot provide appropriate notation, error checking, or optimizations.

[1] A. J. Perlis. Epigrams on programming. *SIGPLAN*, 17(9):7–13, 1982

Domain-specific languages (DSLs) address this problem by providing linguistic abstractions and notations that allow direct and understandable expression of domain concepts instead of encoding these in a lower level programming language. The IDE for the DSL provides editor services such as syntax checking, syntax highlighting, error checking and code navigation. The code generator or interpreter for a DSL encapsulates implementation knowledge, i.e. takes care of the details that are irrelevant to the application developer. Over the last couple of years, interest in DSL has grown significantly as evidenced by the increased uptake of language workbenches such as Eclipse Xtext and mainstream coverage of the subject by authors like Martin Fowler [2].

[2] M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010

While much has been written about tools and techniques for implementing DSLs, little research has gone into the *design* of domain-specific languages. As a result DSL is more an art or craft than an engineering discipline. If DSLs are to become a staple ingredient of software engineering, we need a more systematic engineering approach to the design of domain-specific languages, which requires a well established vocabulary for discussing and *comparing* DSL designs.

1
Core Design Dimensions

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

1.1 *Programs, Languages, Domains*

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

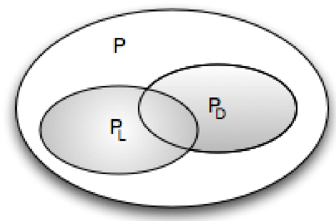

Figure 1.1: Programs, Languages, Domains

NOW, WHAT ARE DOMAINS? There are essentially two approaches to characterize domains. First, domains are often considered as a body of knowledge in the real world, i.e. outside the realm of software. For instance, pension policies are contracts that can be defined and used without software and computers. From this *deductive* or *top-down* perspective, a domain D is a body of knowledge for which we want to provide some form of software support. We define P_D the subset of programs in P that implement computations in D, e.g. 'this program implements a fountain algorithm'.

In the *inductive* or *bottom-up* approach we define a domain in terms of existing software. That is, a domain D is identified as a subset P_D of P, i.e. a set of programs with common characteristics or similar purpose. Often, such domains do not exist outside the realm of software. For example, P_{web} is the domain of web applications, which is intrinsically bound to computers and software. There is a wide variety of programs that we would agree to be web applications. A domain can be very specific. For example P_{vfount} is the set of fountain programs for the particular fountain hardware produced by vendor V. A special case of the inductive approach is where we define a domain as a subset of programs of a specific P_L instead of the more general set P. In this special case we can often clearly identify the commonality between programs in the domain, in the form of their consistent use of a set of domain-specific patterns or idioms.

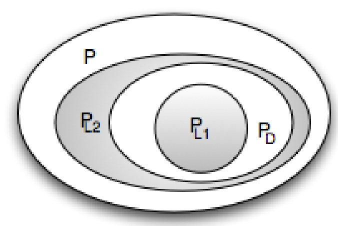

Figure 1.2: Languages L1 and L2 under-approximate and over-approximate domain D.

Whether we take the deductive or inductive route, we can ultimately identify a domain D by a set of programs P_D. There can be multiple languages in which we can express P_D programs. Possibly, P_D can only be partially expressed in a language L (Figure 1.1). A *domain-specific language* L_D for D is a language that is *specialized* to encoding P_D programs. That is, L_D is more efficient in some respect in representing P_D programs. Typically, such a language is *smaller* in the sense that P_{L_D} is a strict subset of P_L for a less specialized language L.

THE CRUCIAL DIFFERENCE BETWEEN LANGUAGES AND DOMAINS is that the former are finitely generated, but that the latter are arbitrary sets of programs the membership of which is determined by a human oracle. This difference defines the difficulty of DSL design: finding regularity in a non-regular domain and capturing it in a language. The resulting DSL provides an explanation or interpretation of the domain, and often requires trade-offs by under- or over-approximation

(Figure 1.2).

1.1.1 Programs as Trees of Elements

Programs are represented in two ways: concrete syntax and abstract syntax. A language definition includes the concrete as well as the abstract syntax, as well as rules for mapping one to the other. *Parser-based* systems map the concrete syntax to the abstract syntax. Users interact with a stream of characters, and a parser derives the abstract syntax by using a grammar. *Projectional* editors go the other way round. User editing gestures directly change the abstract syntax, the concrete syntax being a mere projection that looks (and mostly feels) like text. SDF and Xtext are parser-based, MPS is projectional.

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

While concrete syntax modularization and composition can be a challenge (as discussed in Section 1.8.5), we will illustrate the principles of the composition approaches based on the abstract syntax. The abstract syntax of programs are primarily trees of program *elements*. Every element (except the root) is contained by exactly one parent element. Syntactic nesting of the concrete syntax corresponds to a parent-child relationship in the abstract syntax. There may also be any number of non-containing cross-references between elements, established either directly during editing (in projectional systems) or by a linking phase that follows parsing.

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

A LANGUAGE l defines a set of language concepts C_l and their relationships. We use the term concept to refer to concrete syntax, abstract syntax plus the associated type system rules and constraints as well as some definition of its semantics. In a fragment, each program element e is an instance of a concept c defined in some language l. We define the *concept-of* function co to return the concept of which a program element is an instance: $co \Rightarrow element \rightarrow concept$. Similarly we define the *language-of* function lo to return the language in which a given concept is defined: $lo \Rightarrow concept \rightarrow language$. Finally, we define a *fragment-of* function fo that returns the fragment that contains a given program element: $fo \Rightarrow element \rightarrow fragment$.

We also define the following sets of relations between program elements. Cdn_f is the set of parent-child relationships in a fragment f. Each $c \in C$ has the properties *parent* and *child*. $mathitRefs_f$ is the set of non-containing cross-references between program elements in a fragment f. Each reference r in $Refs_f$ has the properties $from$ and to, which refer to the two ends of the reference relationship. **TODO**▶Do we need those two also for language concepts, or do we always use them with fragments?◀ Finally, we define an inheritance relationship that applies the Liskov Substitution Principle to language concepts. A concept c_sub that extends another concept c_super can be used in places where an instance of c_super is expected. Inh_l is the set of inheritance relationships for a language l. Each $i \in Inh_l$ has the properties $super$ and sub.

An important concept in LMR&C is the notion of independence. An *independent language* does depend on other languages. An independent language l can be defined as a language for which the following hold:

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

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

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

An *independent fragment* is one where all references stay within the fragment (4). By definition, an independent fragment has to be expressed with an independent language (5).

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

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

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

$$\forall e \in E_f \mid lo(e) = l \tag{1.6}$$

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

1.1.2 Domain Hierarchy

The subsetting of domains naturally gives rise to a hierarchy of domains (Fig. 1.3). At the bottom we find the most general domain D_0. It is the domain of all possible programs P. Domains D_n, with $n > 0$, represent progressively more specialized domains, where the set of possible programs is a subset of those in D_{n-1} (abbreviated as D_{-1}). We call D_{+1} a subdomain of D. For example, $D_{1.1}$ could be the domain of embedded software, and D_{1-2} could be the domain of enterprise software. The progressive specialization can be continued ad-infinitum in principle. For example, $D_{2.1.1}$ and $D_{2.1.2}$ are further subdomains of $D_{1.1}$: $D_{2.1.1}$ could be automotive embedded software and $D_{2.1.2}$ could be avionics software. At the top of the hierarchy we find singleton domains that consist of a single program. Languages are typically designed for a particular D. Languages for D_0 are called general-purpose languages. Languages for D_n with $n > 0$ become more domain-specific for growing n.

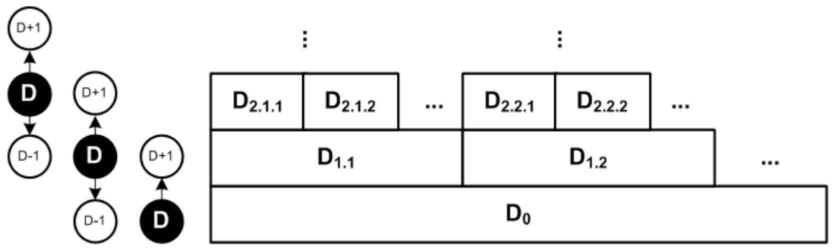

Figure 1.3: Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

1.1.3 Model Purpose

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

Embedded C: The model purpose is the generation of an efficient low-level C implementation of the system, while at the same time providing software developers with meaningful abstractions. ◀

Fountains: The model purpose is the generation of efficient implementation code for various different target platforms. A secondary purpose is enabling domain experts to express the algorithms and experiment with them using simulations and tests. The DSL is not expected to be used to visualize the actual fountain installation or for sales or marketing purposes. ◀

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

1.1.4 Parsing vs. Projection

There are two main approaches for implementing external DSLs. The traditional approach is parser-based. A grammar specifies the sequence of tokens and words that make up a structurally valid program. A parser is generated from this grammar. A parser is a program that recognizes valid programs in their textual form and creates an abstract syntax tree or graph. Analysis tools or generators work with this abstract syntax tree. Users enter programs using the concrete syntax (i.e. character sequences) and programs are also stored in this way. Example tools in this category include Spoofax and Xtext.

Projectional editors (also known as structured editors) work without parsers. Editors directly modify the abstract syntax tree. Projection rules then render a textual (or other) representation of the program. Users read and write programs through this projected notation. Programs are stored as abstract syntax trees, usually as XML. As in parser-based systems, backend tools operate on the abstract syntax tree. Projectional editors are well known from graphical editors, virtually all of them are projectional editors. However, they can also be used for textual syntax. Example tools in this category include the Intentional Domain Workbench (`http://intentsoft.com`) and JetBrains MPS.

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

In this section, we do not discuss the relative advantages and drawbacks of parser-based vs. projectional editors in general. However, we will point out if and when there are different DSL design options depending on which of the two approaches is used.

1.1.5 Design Dimensions

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

1.2 Expressivity

One of the fundamental advantages of domain-specific languages is increased expressivity over more general programming languages. By making assumptions about the domain of application and encapsulating knowledge about the domain in the language and its implementation, e.g. the mapping to a lower level language, programs in the DSL can be significantly more concise by avoiding the need for users to write "boilerplate code". While it is always possible to produce short but incomprehensible programs, overall, shorter programs require less effort to read and write than longer programs, and should therefore be more efficient in software engineering[1]. Thus, we will assume that all other things being equal, shorter programs are preferable over longer programs.

The Kolomogorov complexity [2] of an object is the smallest program in some description language that produces the object. In our case the objects of interest are programs in P and we are interested in designing languages that minimize the size of encodings of programs. We use the notation $|p_L|$ to indicate the size of program p as encoded in language L[3]. The essence is the assumption that, within one language, more complex programs will require larger encodings. We also assume that p_L is the smallest encoding of p in L, i.e. does not contain dead or convoluted code. We can then qualify the expressivity of a language relative to another language.

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

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

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

A weaker version of this statement requires that a language is *mostly* more expressive, but may not be in corner cases.

DSLs ARE MORE EXPRESSIVE than GPLs in the domain they are built for. Higher expressiveness means that less code has to be written to express interesting information within a domain. Generally, shorter programs are better than longer programs. However, there are also other concerns to consider.

Before being able to write these concise programs, users have to learn the language. This can be separated into learning the domain, and learning the syntax of the language. For people who know the domain, learning the syntax can be simplified by using good IDEs with code completion and quick fixes.

TODO▶Talk about small vs. big languages, as a matter of style. modular lang as a tradeoff – see also discussion below.◀

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

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

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

1.2.1 Expressivity and the Domain Hierarchy

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

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

WebDSL: WebDSL entity declarations abstract over the boilerplate code required by the Hibernate framework for annotating Java classes with object-relational mapping annotations. This reduces code size by an order of magnitude [4]. ◀

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

1.2.2 Linguistic Abstraction

By making the concepts of D first class members of a language L_D, i.e. defining linguistic abstractions for these concepts, they can be uniquely identified in a D program and their structure and semantics is well defined. No semantically relevant[5] idioms or patterns are required to express programs interesting in D. Consider the two examples of loops in a Java-like language:

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

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

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

The left loop can be parallelized, since the order of summing up the array elements is irrelevant. The right one cannot, since (we presume) the order of the elements in the created list is relevant. A transformation engine that translates and optimizes the programs, must perform (sophisticated, and sometimes impossible) program analysis to determine that the left loop can be parallelized. The following alternative expression of the same behaviour uses better linguistic abstractions, because it is clear without analysis that the first loop can be parallelized and the second cannot[6]. The decision can simply be based on the language concept used (*for* vs. *seqfor*)

[6] The property of a language L_D of having first-class concepts for abstractions relevant in D is often called declarative: 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.

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

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

Linguistic abstraction also means that no details irrelevant to the model purpose are expressed. Once again, this increases conciseness, and avoids the undesired specification of unintended semantics (over-specification). In the example above, the loop A is over-specified, it expresses ordering of the operations, although this is (most likely) not intended by the person who wrote the code.

Embedded C: State machines can be implemented as switch/case blocks or as arrays pointing into each other. The DSL program does not specify which implementation should be used and the transformation engine is free to chose the more appropriate representation. Also, log statements and task declarations can be translated in two different ways depending on the target environment. ◀

1.2.3 *In-Language Abstraction*

Conciseness can also be achieved by a language providing facilities to define new (non-linguistic) abstractions in programs. It is *not* a sign of a bad DSL if it has in-language abstraction mechanisms as long as the created abstractions don't require special treatment by analysis or processing tools — at which point they should be refactored into linguistic abstractions.

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

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

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

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

1.2.4 Standard Library

If a language provides support for in-language abstraction, these facilities can be used by the language *designer* to provide functionality to language users. Instead of adding language features, a standard library is deployed along with the language. This approach keeps the language itself small, and allows subsequent extensions of the library without changing the language definition and processing tools. TODO▶Example from the fridge language◀

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

1.2.5 Linguistic vs. In-Language Abstraction

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

Using a standard library may be a good compromise where one set of users develops the abstractions to be used by another set of developers. This is especially useful if the same language should be used for several, related projects or user groups. Each can build their own set of abstractions in the library. It should be kept in mind that in-language abstraction only works if the transformation of these abstractions is not specific to the abstraction. In that case, linguistic abstraction is better suited.

Modular language extension, as discussed later, provides a middle ground between the two approaches,

1.3 Coverage

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

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

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

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

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

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

IDEALLY, A DSL WILL COVER ALL OF ITS DOMAIN ($C_D(L_D)$ is 100%). There can be two reasons for a DSL *not* to cover all of its *own* domain D. First, the language may be deficient and needs to be redesigned. This is especially likely for new and immature DSLs. Scoping the domain for which to build a DSL is an important part of DSL design. Second, the language may have been defined expressly to cover only a subset of D, typically the subset that is most often used. Covering all of D may lead to a language that is too complicated for the intended user community. This requires coordination between DSL users and D_{-1} users, if this not the same group of people. However, after this coordination is established it is easier to cope with unsupported cases resulting from a DSL that has not yet been updated to an evolving domain.

In this case, the remaining parts of D may have to be expressed with code written in D_{-1}.

Full coverage is desirable. It requires, however, that the domain is well-defined and we can actually know what full coverage is. Also, over time, it is likely that the domain evolves and grows, and the language has to be continuously evolved to keep coverage full. Full coverage may lead to a language that has support for rarely used corner cases in the domain, increasing the size of the language. This may make the language unsuitable for (the majority of the) domain users.

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

> **WebDSL:** WebDSL defines web pages through 'page definitions' which have formal parameters (Figure **??**). 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. ◀

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

1.4 Semantics and Execution

Semantics can be partitioned into static semantics and execution semantics. Static semantics represent the constraints and type system rules. Execution semantics denote the observable behaviour of a program p as it is executed. In this section we focus on execution semantics. Using a function OB that defines this observable behaviour we can define the semantics of a program p_{L_D} by mapping it to a program q in a language for D_{-1} that has the same observable behavior:

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

Equality of the two observable behaviors can be established with a sufficient number of tests, or with model checking and proof in rare cases. Details are beyond the scope of this paper. This definition reflects the hierarchy of domains and works both for languages that describe only structure, as well as for those that include behavioural definitions. The technical implementation of the mapping to D_{-1} can be provided in two different ways: a DSL program can literally be transformed into a program, or a L_{D-1} an interpreter can be written in L_{D-1} or L_{D_0} to execute the program. A complete discussion of the relative merits between transformation and interpretation is beyond the scope of this paper.

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

1.4.1 Transformation

The transformation case is easy to see: a transformation recreates those patterns and idioms in L_{D-1} for which L_D provides linguistic abstraction. The result may be transformed further, until a level is reached for which a language with an execution infrastructure exists — often D_0. Code generation from a DSL is thus a special case where L_{D_0} code is generated.

Embedded C: The semantics of state machines are defined by their mapping back to C switch-case statements. This is repeated for higher D languages. The semantics of the robot control DSL is defined by its mapping to state machines and tasks. To explain the semantics to the users, prose documentation is available as well. ◀

For languages where the semantic gap between the DSL and the target language is significant, it makes sense to introduce intermediate languages so the transformation can be modularized. Optimizations can be performed on each of these levels, exploiting the properties of that particular abstraction level. Modern compilers with multiple intermediate representations are a good example for this approach. MPS comes with a transformation engine that makes such transformation chains easy to implement.

We can learn something else from compilers: they can be retargetted relatively easily by exchanging the backends (machine code generation phases) or the frontend (programming language parsers and analyzers). For example, GCC can generate code for many different processor architectures (exchangeable backends), and it can generate backend code for several programming languages, among them C, C++ and Ada (exchangeable frontends). The same is possible for DSLs. The same high D models can be executed differently by exchanging the lower D intermediate languages and transformations. Or the same lower D languages and transformations can be used for different higher D languages, by mapping these different languages to the same intermediate language.

Embedded C: The embedded C language (and some of its higher D extensions) have various translation options, for several different target platforms (Win32 and Osek), an example of backend reuse. ◀

Reusing lower D languages and their subsequent transformations includes reuse of non-trivial analyses or optimizations. This makes this case much more useful than what the pure reuse of languages and transformations suggests. This is the reason why we usually generate GPL source code from DSLs, and not machine code: we want to reuse existing transformations and optimizations provided by the GPL compiler and/or runtime.

Splitting a transformation into several subsequent steps is useful to be able to reuse some of these steps. Frontends and backends can be reused, for example supporting different target platforms. This is especially useful if the reusable transformations include non-trivial optimizations, or are just big (i.e. generate many different artifacts, fan-out). Splitting a transformation into a chain of smaller ones also makes each of them easier to understand and maintain.

1.4.2 Interpretation

For interpretation, the same approach could be used, i.e. an interpreter for L_D can be implemented in L_{D-1}. However, in practice we see interpreters written in L_{D_0}. They are extensible, so new interpreter code can be added in case specialized languages define new language concepts. We have not come across real-world DSLs where interpreters are stacked along the domain hierarchy in the same way as transformations, with interpreters for L_{D_n} written in $L_{D_{n-1}}$ for $n > 1$.

Fountains: The DSL also supports the definition of unit tests for the asynchronous, reactive pumping 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 pumping program, stepping through it in single steps, watching values change. ◀

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

1.4.3 Definition of Semantics

Defining semantics in practice happens by mapping the DSL concepts to D-1 concepts for which the semantics is known. For DSLs used by developers, and for domains that are defined bottom-up, this works well. For domain expert DSLs, and for domains define top-down this approach is not necessarily good enough, since because the D-1 concepts has no inherent meaning to the users and/or the domain. An additional way of defining the meaning of the DSL is required. Useful approaches include prose documentation as well as test cases. These can be written in (another part of the) DSL itself; this way, domain users can play with the DSL and write down their expectations in the testing aspect.

1.4.4 *Transformation vs. Interpretation*

The primary concern in semantics is the decision between transformation (code generation) and interpretation. We present a couple of criteria to help with the 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 D-1. Interpreters are meta programs and as such harder to relate to the existing code patterns.

Debugging Debugging generated code is straight forward if the code is well structured (which is up to the transformation). Debugging interpreters is harder, because setting breakpoints in the DSL program requires the use of conditional breakpoints, which are typically cumbersome to use.

Performance and Optimization The code generator can include 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. Generally, performance is better in generated environments, since interpreters always imply an additional layer of indirection.

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

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

Runtime Change In interpreted environments, the DSL program can be changed as the target system runs; the editor can be integrated into the executing system. The term data-driven system is often used in this case.

1.4.5 *Sufficiency*

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

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

Embedded C: The MED code is sufficient regarding the MED-to-C generator. It is not sufficient regarding the MED-to-Osek generator, since a so-called *system specification* fragment is required to define how the mapping to Osek should look like. ◀

1.4.6 Multiple Mappings

The approach suggested so far works well if we have only one mapping of a DSL for execution. The semantics implied by the mapping to L_{D-1} can be *defined* to be correct. However, as soon as we transform the program to several different languages in D_{-1} using several transformations, we have to ensure that the semantics of all resulting programs are identical. In practice, this often happens when 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. In this case, we recommend providing a set of test cases that are executed both the interpreted and generated versions, expecting them to succeed in both. If the coverage of these test cases is high enough to cover all of the observable behavior, then it can be assumed with reasonable certainty that the semantics are the same.

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. ◀

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

- Heuristics, based on patterns and idioms used in the program, can be used to determine the applicable translation from the program.

- In analogy to compiler switches, the decision can be controlled by additional, external data, for example by adding an annotation model. An annotation model contains data used by the transformation to decide how to translate the core program. The transformation uses the L_D program and the annotation model as its input.

- Alternatively, L_D can be extended to contain additional data to guide decision.

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

Mapping the same DSL to different D-1 languages or to different programs expressed with the same D-1 language is useful if different non-functional concerns should be implemented, if several target platforms should be supported, or if several alternative artifacts (code, reports, visualizations) should be selectable. Determining the mapping can be done in several ways: the language can be extended to include explicit information to guide the decision. This is only useful if the DSL user can actually decide which alternative to choose, and if only one alternative should be chose for each program. An alternative is to put the explicit information into a an annotation model (a separate viewpoint), which leads to all the advantages of viewpoints generally (see next section). Specifically, there can be several different annotations for the same code model. A third alternative is to use rules or heuristics, that determine by inspecting the model which translation to use. Codifying these rules and heuristics can be hard.

1.4.7 Reduced Expressiveness

It may be beneficial to limit the expressiveness of a language. Limited expressiveness often results in more sophisticated analyzability. For example, while state machines are not very expressive, sophisticated model checking algorithms are available (e.g. using the SPIN model checker from `http://spinroot.com/`). The same is true for first-order logic, where satisfiability (SAT) solvers [7] can be used to check programs for consistency. If these kinds of analysis are useful for the model purpose, then limiting the expressiveness to the respective formalism may be a good idea, even if it makes expressing certain programs in D more cumbersome. An alternative approach is to use a language with limited expressiveness at D_{-1}. For analysis and verification, the L_D concepts are translated down into the verifiable L_{D-1} language. Verification is performed on L_{D-1}, mapping the results back to L_D. Transforming to a verifiable formalism also works if the formalism is not at D_{-1}, as long as a mapping exists, and as long as the results of the verification can be mapped back to the DSL program.

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

Fountains: For the pumping algorithm, we are working on a mapping to SPIN/Promela formalism to perform model checking on the pumping algorithms. This will be used to proof that invariants expressed in the pumping program will hold in all cases, or to show counter examples if they do not. ◀

Reduced expressiveness may enhance the ability to perform meaningful verifications. Examples include model checking for state machines or the use of solvers for first order logic. However, to enable these approaches, 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. In this case a DSL may be translated to the formal language for verification. The problem with this appraoch is the back-translation of analysis results to the DSL. Domain users will not be able to interpret the results of model checkers or solvers, they have to be translated back to the DSL. This may be a lot of work, or even impossible.

1.5 Separation of Concerns

A domain D can be composed from different concerns. Fig. 1.4 shows $D_{1.1}$ composed from the concerns A, B and C. To describe a complete program for D, the program needs to address all the concerns.

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

Two fundamentally different approaches are possible to deal with the set of concerns in a domain. Either a single, integrated language can be designed that addresses all concerns of D in one integrated model. Alternatively, separate concern-specific DSLs can be defined, each addressing one or more of the domain's concerns. A program then consists of a set of dependent, concern-specific fragments, that relate to each other in a well-defined way. Viewpoints support this separation of domain concerns into separate DSL. Fig. 1.5 illustrates the two different approaches.

Embedded C: The tasks language module includes the task implementation as well as task scheduling in one language construct. Scheduling and implementation are two concerns that could have been separated. We opted against this, because both concerns are specified by the same person. The language used for implemen-

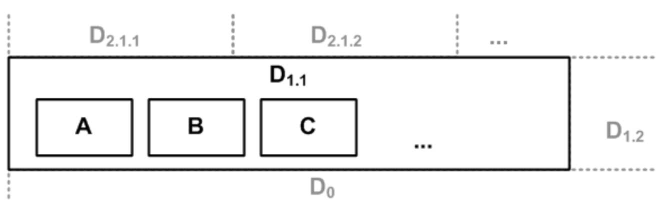

Figure 1.4: A domain may consist of several concerns. A domain is covered either by a DSL that addresses all of these concerns, or by a set of related, concern-specific DSLs.

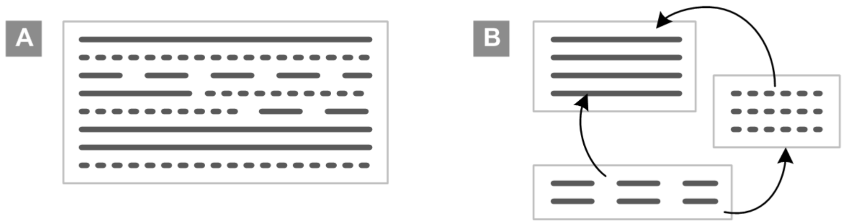

Figure 1.5: Part A shows an integrated DSL, where the various concerns (represented by different line styles) are covered by a single integrated language (and consequently, one model). Part B shows several viewpoint languages (and model fragments), each covering a single concern. Arrows in Part B highlight dependencies between the viewpoints.

tation code is med.core, whereas the task constructs are defined in the med.tasks module. So the languages are modularized, but they are used in a single integrated model. ◀

WebDSL: Web programs consists of multiple concerns including persistent data, user interface, and access control. WebDSL provides specific languages for these concerns, but *linguistically integrates* them into a single language [8]. 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. ◀

[8] 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

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

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

SEPARATING OUT A DOMAIN CONCERN into a separate viewpoint fragment can be useful for several reasons.

If different concerns of a domain are specified by different stakeholders then separate viewpoints make sure that each stakeholder has to deal only with the information they care about. The viewpoint separation has to be aligned with the development process: the order of creation of the fragments must be aligned with the dependency structure.

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

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 enable this.

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.

Fountains: One concern in this DSL specifies the hardware structure of fountain installations. The other one describes the fountain pumping algorithm. Both are implemented as separate viewpoints, where the algorithm DSL references the hardware structure DSL. Using this dependency structure, different algorithms can be defined for the same structure. Each of these algorithms resides in its own model. While the C code generation requires both behavior and hardware structure fragments, the hardware fragment is sufficient for a transformation that creates a visual representation of the hardware structures. ◀

1.5.1 Viewpoint Synchronization

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

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

1.5.2 Views on Programs

In projectional editors it is also possible to store the data for all viewpoints in the same model, while showing different "views" onto the model to materialize the various viewpoints. The particular benefit of this approach is that additional concern-specific views can be defined later, after programs have been created. MPS also provides so-called annotations, where additional model data can be "attached" to any model element, and shown optionally.

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

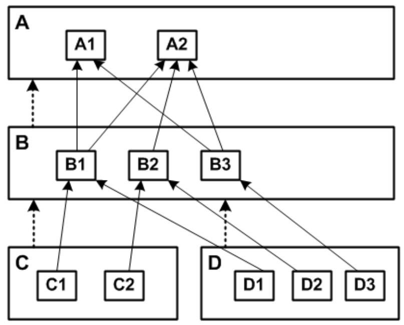

Figure 1.6: Progressive refinement: the boxes represent models expressed with corresponding languages. The dotted arrows express dependencies, whereas the solid arrows represent references between model elements.

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

1.5.3 *Viewpoints for Progressive Refinement*

There is an additional use case for viewpoint models not related to the concerns of a domain, but to progressive refinement. Consider complex systems. Development starts with requirements, proceeds to high-level component design and specification of non-functional properties, and finishes with the implementation of the components. Each of these refinement steps may be expressed with a suitable DSL, realizing the various "refinement viewpoints" of the system (Fig. 1.6). The references between model elements are called traces[9]. Since the same conceptual elements may be represented on different refinement levels (e.g. component design and component implementation), synchronization between the viewpoint models is often required (enabled via techniques described in [10]).

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

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

1.6 *Completeness*

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

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

guages at D_0 are by definition complete.

Embedded C: The Embedded C language is complete regarding D_{-1}, or even D-n for higher levels of D, since higher levels are always built as extensions of D_{-1}. Developers can always fall back to D_{-1} to express what is not expressible in D. ◀

Fountains: This DSL uses several viewpoints: one to define the structure of fountain installations, and one to describe the actual algorithm. When generating the C code for the algorithm, both viewpoints are needed, because the structure defines how some of the algorithms are implemented. ◀

A complete DSL is capable of specifying the complete semantics of a program in domain D. Possibly, several viewpoints are used to allow different stakeholders to express their concerns undisturbed by other concerns. In fact, the manually written D_{-1} code can be considered just another viewpoint, which is expressed in a GPL, and implemented by developers.

For DSLs used by developers, incomplete DSLs are usually not a problem because they are comfortable with providing the D_{-1} code expressed in a programming language. Specifically, the DSL users are the same people as those who provide the remaining D_{-1} code, so co-ordination between the two not a problem. For DSLs used by domain experts, the situation is different. Usually, they are not able to write D_{-1} code, so other people (developers) have to fill in the remaining concerns.

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

1.6.1 *Compensating for Incompleteness*

Integrating the L_{D-1} in case of an incomplete L_D language can be done by calling "black box" code written in L_{D-1} (requires concepts for calling D_{-1} foreign functions in the D language), by directly embedding L_{D-1} code in the L_D program (especially useful if L_D is an extension of L_{D-1}), by inserting manually-written L_{D-1} code into the L_{D-1} code generated from the L_D program (using protected regions), or by using composition mechanisms of L_{D-1} (e.g. inheritance) to "plug in" the manually written code into the generated code without actually modifying the generated files (also known as the Generation Gap pattern [11]).

[11] J. Vlissidis. Generation gap. C++ Report, 1996

Compensating for incompleteness can be done is several ways. Manually written code can be directly embedded into the DSL program at the respective locations. This is useful if the tool provide adequate support for embedding the D-1 language into D programs.Just "pasting text into a textfield" is not a productive approach, since no real integration between the languages is supported. If tool support is available, then this approach is useful only if the users of the DSL

Many tools for graphical modeling provide small boxes in dialogs to enter "target code" without any support. In my opinion, this is not useful!

are also able to write the D-1 code. If this is not the case, one of the two alternative approaches is indicated. The first one is the use of the generation gap pattern **TODO**►cite◄ where the D-1 code is composed with the code generated from the DSL. Alternatively, developers can develop a predefined set of foreign functions that can be called from within the DSL. In effect, developers provide a library of "behavioural building blocks" which can be invoked as black boxes from DSL programs.

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. ◄

Note that a DSL that does not *cover* all of D can still be *complete*: the L_{D-1} code generated from a D program may require a framework written in L_{D-1} to run in. That framework represents aspects of D outside the scope of L_D.

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

Embedded C: Control loops and device drivers are outside the scope of the DSL, but are used by the generated code. These parts are manually written and made available to the generated code by well-defined APIs. ◄

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

1.6.2 Roundtrip Transformation

Roundtrip transformation means that an L_D program can be recovered from a program in L_{D-1} (written from scratch, or changed manually after generation from a previous iteration of the L_D program). This is challenging, because it requires reconstituting the semantics of the L_D program from idioms or patterns used in the L_{D-1} code. This is the general reverse engineering problem and is not generally possible, although progress has been made over recent years (see for example [12]). For complete languages roundtripping is generally not useful, because the complete program can be written on L_D in the first place. Even if recovery of the semantics is possible it may not be practical: if the DSL provides significant abstraction over the L_{D-1} program, then the generated L_{D-1} program is so complicated, that manually changing the D_{-1} code in a consistent and correct way is tedious and error-prone.

[12] D. Beyer, T. A. Henzinger, and G. ThÃl'oduloz. Program analysis with dynamic precision adjustment. In *ASE*, pages 29–38, 2008; M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBMSJ*, 46(2):265–288, 2007; and M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Fast extraction of high-quality framework-specific models from application code. *ASE*, 16(1):101–144, 2009

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

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

Roundtripping has traditionally been used with respect to UML models and generated class skeletons. In that case, the abstractions were similar (classes), it the tool basically just provides a different concrete syntax.

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

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

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

1.7 *Structuring Programs*

Languages have to provide means to structure large models in order to keep them manageable. These include modularization and visibility, specification vs. implementation, specialization, types and instances as well as partitioning.

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

1.7.1 *Modularization and Visibility*

DSL often provide some kind of logical unit structure, such as namespaces or modules. Visibility of symbols may be restricted to those in the same unit, or in referenced ("imported") units. Symbols may be declared as public or private. Most contemporary programming languages use some form of namespaces and visibility restriction as their top level structure.

Some form of namespaces and visibility is necessary in almost any DSL. Often there are domain concepts that can play the role of the module, possibly oriented towards the structure of the organization in which the DSL is used. Keeping some elements private makes those changeable without consequences for using modules.

Embedded C: As a fundamental extension to C, this DSL contains modules with visibility specifications and imports. Functions, state machines, tasks and all other top-level concepts reside in modules. ◄

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

1.7.2 *Partitioning*

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

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

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

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

Three main forces drive the use of partitions. One is the scalability of the DSL tool. Beyond a certain file size, the editor may become sluggish. The second reason is to use partitions as a way of organizing the overall model. This is particularly important if the tool does not provide a good means of showing the logical structure models and finding elements by name and type. Organizing files, with useful names, in directory structures is a good alternative. The third driver for partitioning is integration with existing, file based tool chains. Files may be the unit of checkin/checkout, versioning, branching or permission checking.

1.7.3 Specification vs. Implementation

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

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

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

Fountains: Pumping programs can refer to entities defined as part of the fountain hardware as a means of accessing hardware elements (pumps, vents, switches). To enable pumping programs to run with different, but similar hardware configurations, the hardware structure can use "trait inheritance", where a hardware trait defines a set of hardware elements. Other fountain configurations can inherit these traits. As long as pumping programs are only written against traits, they work with any fountain that implements the particular set of traits against which the program is written. ◀

1.7.4 *Specialization*

Specialization enables one entity to be a more specific variant of another one. Typically, the more specific one can be used in all context where the more general one is valid (Liskov substitution principle[13]). The more general one may be incomplete, requiring the definition of specialized ones that "fill in the holes" left undefined by the more general one. Specialization in the context of DSLs can be used for implementing variants or of evolving a model over time.

[13] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994

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

Pension Plans: The customer had to create a huge set of pension plans, implementing changes in relevant law over time, or for slightly different customer groups. Copying complete plans and then making adaptations was not feasible. Hence the DSL provides a way for pension plans to inherit from one another. Calculation rules can be marked *abstract* (requiring overwriting in sub-plans), *final* rules are not overwritable. Visibility modifiers control which rules are considered "implementation details". ◀

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

1.7.5 *Types and Instances*

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

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

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

1.7.6 *Superposition and Aspects*

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

This is especially important in the context of product line engineering and is discussed in

M. Voelter and E. Visser. Product line engineering using domain-specific languages. In *Software Product Line Conference*, 2011

Embedded C: This DSL provides a way of advising component definitions, to introduce new ports from within an aspect. This is used, for example, to modularize the Monitoring concern. ◀

WebDSL: Entity declarations can be *extended* in separate modules.

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

1.8 Language Modularity

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

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

We have identified the following four: referencing, extension, reuse and embedding. We distinguish them regarding fragment structure and language dependencies, as illustrated in Fig. 1.7. Fig. 1.8 shows the relationships between fragments and languages in these cases. We consider these two criteria to be essential for the following reasons.

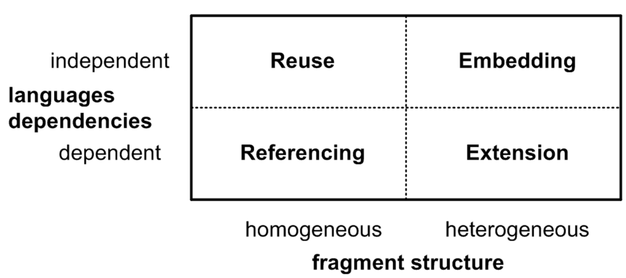

Figure 1.7: We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies.

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

1.8.1 Language Referencing

Language referencing (Fig. 1.9) enables *homogeneous* fragments with cross-references among them, using *dependent* languages.

A fragment f_2 depends on f_1. f_2 and f_1 are expressed with different languages l_2 and l_1. The referencing language l_2 depends on the

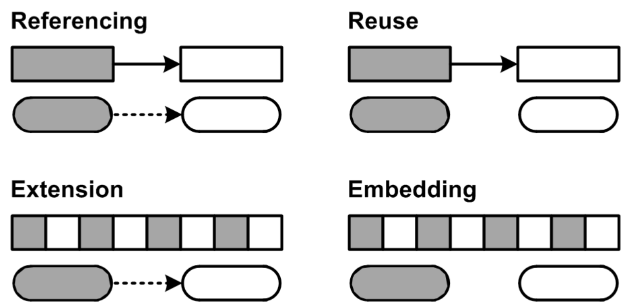

Figure 1.8: The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

referenced language l_1 because at least one concept in the l_2 references a concept from l_1. We call l_2 the *referencing* language, and l_1 the *referenced* language. While equations (2) and (3) continue to hold, (1) does not. Instead

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

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

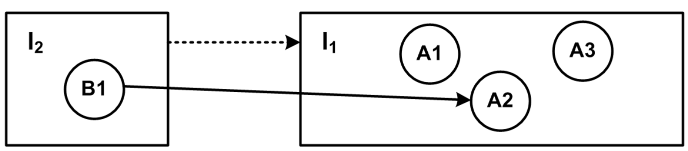

Figure 1.9: Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

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

Fountains: As an example, consider the domain of refrigerator configuration. The domain consists of four concerns. The first concern H describes the hardware structure of refrigerators appliances including compartments, compressors, fans, vents and thermometers. The second concern A describes the cooling algorithm using a state-based, asynchronous language. Cooling programs refer to hardware building blocks and access the their properties in expressions and commands. The third concern is testing

T. A cooling test can test and simulate cooling programs. The fourth concern P is parametrization, used to configure a specific hardware and algorithm with different settings, such as the actual target cooling temperature. The dependencies are as follows: $A \rightarrow H, T \rightarrow A, P \rightarrow A$. Each of these concerns are implemented as a separate language with references between them. H and A are separated because H is defined by product management, whereas A is defined by thermodynamicists. Also, several algorithms for the same hardware must be supported, which makes separate fragments for H and A useful. T is separate from A because tests are not strictly part of the product definition and may be enhanced after a product has been released. Finally, P is separate, because the parameters have to be changed by technicians in the field, and several parametrizations for the same algorithm exist. These languages have been built as part of a single project, so the dependencies between them are not a problem. ◀

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

Progressive Refinement Progressive refinement, also introduced earlier, also makes use of language referencing.

1.8.2 Language Extension

Language extension Fig. 1.9 enables *heterogeneous* fragments with *dependent* languages. A language l_2 extending l_1 adds additional language concepts to those of l_1. We call l_2 the *extending* language, and l_1 the *base* language. To allow the new concepts to be used in the context provided by l_1, some of them extend concepts in l_1. So, while l_1 remains independent, l_2 becomes dependent on l_1 since

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

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

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

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

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

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

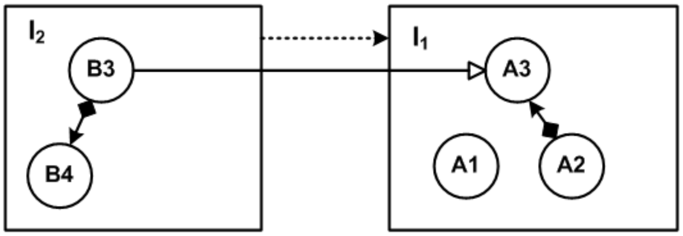

Figure 1.10: Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

TODO▶Wrong picture! Create new Screenshot from Visio!◀

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

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

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

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

Extension is especially useful for bottom-up domains. The common patterns and idioms identified for a domain can be reified directly into linguistic abstractions, and used directly in the language from which they have been embedded. Incomplete languages are not a problem, since users can easily fall back to D_{-1} to implement the rest. Since

DSL users see the D_{-1} code all the time anyway, they will be comfortable falling back to D_{-1} in exceptional cases. This makes extensions suitable only for DSLs used by developers. Domain expert DSLs are typically not implemented as extensions.

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

1.8.3 Language Reuse

Language reuse (Fig. 1.11) enables *homogenous* fragments with *independent* languages. Given are two independent languages l_2 and l_1 and two fragment f_2 and f_1. f_2 depends on f_1, so that

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

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

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

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

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

Language referencing supports reuse of the referenced language. Language reuse supports the reuse of the *referencing* language as well. Consider as examples a language for describing user interfaces. It provides language concepts for various widgets, layout definition and disable/enable strategies. It also supports data binding, where data structures are associated with widgets, to enable two-way synchronization between the UI and the data. Using language reuse, the same UI language can be used with different data description languages. Referencing is not enough because the UI language would have a direct dependency on a particular data description language. Changing

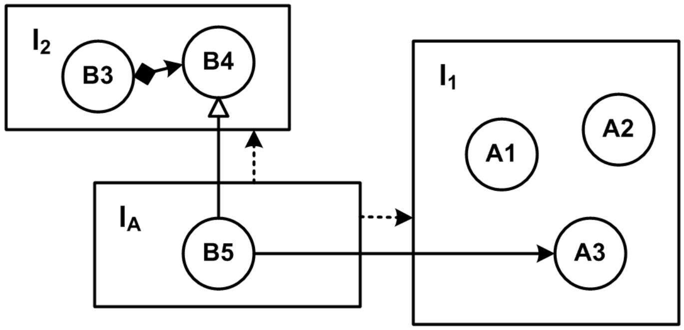

Figure 1.11: Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

the dependency direction to $data \rightarrow ui$ doesn't solve the problem either, because this would go against the generally accepted idiom that UI has dependencies to the data, but not vice versa (**TODO**►cite mvc◄).

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

Reuse supports the case where all participating languages have no dependencies on the other languages and can be reused independently. This makes sense for concern DSLs that have the potential to be reused in many domains, with minor adjustments. Examples include role-based access control, relational database mappings.

1.8.4 Language Embedding

Language embedding (Fig. 1.12) enables *heterogeneous* fragments with *independent* languages. It is similar to reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now embed instances of concepts from l_2 in a fragment f expressed with l_1, so

$$\begin{aligned} \forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge \\ lo(co(c.child)) = (l_1 \vee l_2) \end{aligned} \tag{1.14}$$

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

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

Embedding supports syntactic composition of independently developed languages. As an example, consider a state machine language

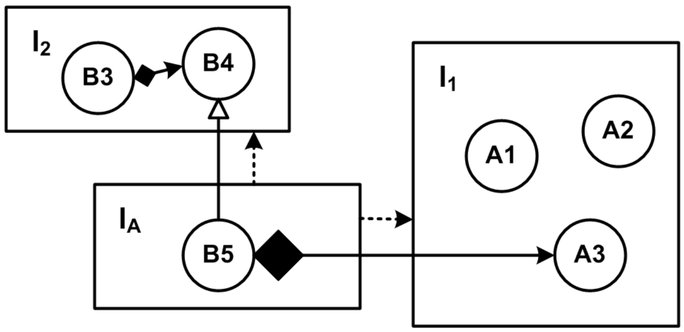

Figure 1.12: Embedding: l_1 and l_2 are independent languages. However, we still want to use them in the same fragment. To enable this, an adapter language l_A is added. It depends on both l_1 and l_2, and uses inheritance and composition to adapt l_1 to l_2 (this is the almost the same structure as in the case of reuse; the difference is that $B5$ now contains $A3$, instead of just referencing it.)

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

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

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

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

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

1.8.5 Implementation Challenges and Solutions

Syntax Referencing and Reuse keeps fragments homogeneous. Mixing of concrete syntax is not required. A reference between fragments is usually simply an identifier and does not have its own internal structure for which a grammar would be required. The name resolution phase can then create the actual cross-reference between abstract syntax objects. In the refrigerator example, the algorithm language contains cross-references into the hardware language. Those references are simple, dotted names (*a.b.c*). In the UI example, the adapter language simply introduces dotted names to refer to fields of data structures.

Extension and Embedding requires modular concrete syntax definitions because additional language elements must be "mixed" with programs written with the base language. In the embedded programming example, state machines are hosted in regular C programs. This works because the C language's `Module` construct contains a collection of `ModuleContents`, and the `StateMachine` concept extends the `ModuleContent` concept. This state machine language is designed specifically for being embedded into C, so it can access and extend the `ModuleContent` concept. If the state machine language were reusable with any host language in addition to C, then an adapter language would provide a language concept that adapts C's `IModuleContent` to `StateMachine`, because `StateMachine` cannot directly extend `IModuleContent` — it does now depend on the C language.

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

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

For reuse and embedding, the typing rules that affect the interplay between the two languages reside in the adapter language. In the UI example the adapter language will have to adapt the data types of the fields in the data description to the types the UI widgets expect. For example, a combo box widget can only be bound to fields that have some kind of text or enum data type. Since the specific types are specific to the data description language (which is unknown at the time of creation of the UI language), a mapping must be provided in the adapter language.

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

TODO▶create better versions of the margin figures from the Visio diagrams◀

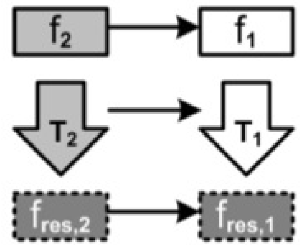

Figure 1.13: Referencing: Two separate, dependent, single-source transformations

Three cases have to be considered for referencing. The first one (Fig. 1.13) propagates the referencing structure to two independent target fragments. We call these two transformations single-sourced, since each of them only uses a single, homogeneous fragment as input and creates a single, homogeneous fragment, perhaps with references between them. Since the referencing language is created with the knowledge about the referenced language, the generator for the referencing language can be written with knowledge about the names of the elements that have to be referenced in the other fragment. If a generator for the referenced language already exists, it can be reused unchanged. The two generators basically share naming information.

> **Fountains:** The refrigerator example uses this case. From the hardware description we generate an XML file that configures a framework that actually collects the data behind the properties of the hardware components in the running refrigerator. The C code generated from the algorithm accesses that framework, using agreed-upon identifiers to identify properties whose value have to be read or set. ◀

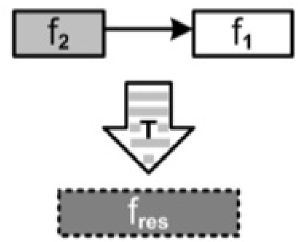

Figure 1.14: A single multi-sourced transformation.

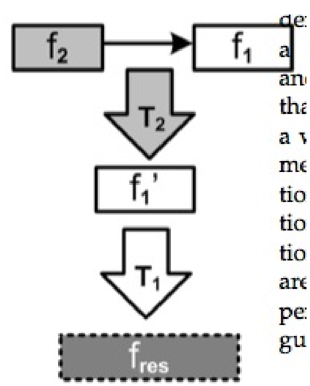

Figure 1.15: A preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment

The second case (Fig. 1.14) is a multi-sourced transformation that creates one single homogeneous fragment. This typically occurs if the referencing fragment is used to guide the transformation of the referenced fragment, for example by specifying target transformation strategies. In this case, a new transformation has to be written that takes the referencing fragment into account. The possibly existing generator for the referenced language cannot be reused as is. An alternative to rewriting the generator is the use of a preprocessing transformation (Fig. 1.15), that changes the referenced fragment in a way consistent with what the referenced fragment prescribes. The existing transformations for the referenced fragment can then be reused.

As we have discussed above, language extensions are usually created by defining linguistic abstractions for common idioms of a domain D. A generator for the new language concepts can simply recreate those idioms when mapping L_D to L_{D-1}, a process called assimilation. In other words, transformations for language extensions map a heterogeneous fragment (containing L_{D-1} and L_D code) to a homogeneous fragment that contains only L_{D-1} code (Fig. 1.16). In some cases additional files may be generated, often configuration files. In the embedded systems state machines example, the state machines are generated down to a function that contains a switch/case statement, as well as enums for states and events.

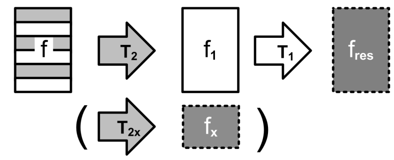

Figure 1.16: Extension: transformation usually happens by assimiliation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often some configuration files.

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

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. Consider the (somewhat constructed) example of two extensions to Java that each define a new statement. When assimilated to pure Java, both new statements require the surrounding Java class to extend a specific, but different base class. This won't work because a Java class can only extend one base class. Interactions may also be more subtle and affect memory usage or execution performance. Note that this problem is not specific to languages, it can occur whenever several independent extensions of a base concept can be used together, ad hoc. To avoid the problem, transformations should be built in a way so that they do not "consume scarce resources" such as inheritance links. A more thorough discussion of the problem of semantic interactions is beyond the scope of this paper, and we refer to [14] for details.

[14]

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.

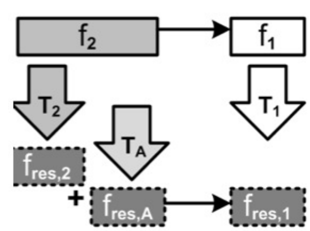

Figure 1.17: Reuse: Reuse of existing transformations for both fragments plus generation of adapter code

There are three cases to consider. The first one, illustrated in Fig. 1.17, describes the case where there is an existing transformation for the reused fragment and an existing transformation for the context fragment — the latter being written with the knowledge that later extension will be necessary. In this case, the generator for the adapter language may "fill in the holes" left by the reusable generator for the referencing language For example, the generator of the context language may generate a class with abstract methods; the adapter may generate a subclass and implement these abstract methods.

In the second case, Fig. 1.18, the existing generator for the reused fragment has to be enhanced with transformation code specific to the context language. In this case, a mechanism for composing transformations is needed.

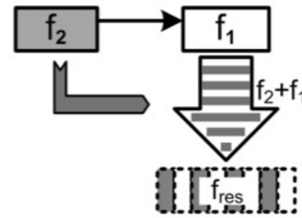

Figure 1.18: Reuse: composing transformations

The third case, Fig. 1.19, leaves composition to the target languages. We generate three different independent, homogeneous fragments, and a some kind of weaver composes them into one final, heterogeneous artifact. Often, the weaving specification is the intermediate result generated from the adapter language. An example implementation could use AspectJ.

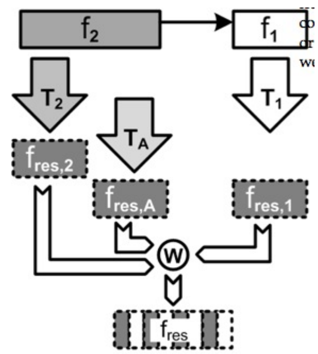

Figure 1.19: Reuse: generating separate artifacts plus a weaving specification

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

If the embeddable language comes with a generator that transforms to the same target language as the embedding language, then the generator for the adapter language can coordinate the two, and make sure a single, consistent fragment is generated. Fig. 1.20 illustrates this case.

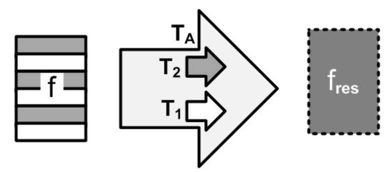

Figure 1.20: In transforming embedded languages, a new transformation has to be written if the embedded language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the emebedded languages.

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

1.8.6 Tool Support

Eclipse Xtext Regarding syntax, language referencing is supported by Xtext. The grammar language comes with special support to reference concepts defined in other languages (or Ecore-based meta model in general). Language extension is also supported, but for at most one base language. Consequently, language reuse is possible as well. However language embedding is not supported because of the limitation to extend only one other language.

Constraints are implemented in Xtext as Java methods that query the AST and report errors. Xtext does not directly support a concise definition of type systems, but third party solutions exist, (e.g. the Xtext Typesystem Framework[15]. Xtext also comes with XBase, a reusable expression language. Languages can extend Xbase to reuse these expressions. Xbase comes with a framework for type calculations. This framework can be extended in a principled way to integrate the typing rules of a language that extends XBase.

[15] http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/

Xtext comes with a transformation and code generation language called Xtend. It supports dependency injection as a first class citizen. This makes it possible to override parts of generators selectively, if the original generator developer has delegated the relevant parts to injected classes. This facility supports the transformation challenges outlined in the previous section nicely.

JetBrains MPS MPS supports syntax composition. Since no grammar is used, the extending language simply creates new language constructs, some of them extending concepts from the base language. In cases where ambiguity would arise in a grammar-based system, MPS requires the language user to decide during input which of the alternatives should be used. A language in MPS can extend any number of languages. In particular, for a given base language, a program can be written using any number of languages extending this base language, without first defining a composed language explicitly. For example, a set of extensions to C can be developed independently, and users can then decide ad hoc which extensions to use in a program (as long as the extensions are semantically compatible). From a syntactic perspective, all LMR&C approaches are supported by MPS without any limits.

MPS also supports restriction. A language concept can restrict under which parents it can be used, or which children a concept may have. These constraints literally remove the ability to enter constructs that would violate the constraints. Using this approach, the unit testing extension to C enforces that `assert` statements can only be used in unit tests and not in any other statement list.

MPS has another approach for language embedding called *attributes*. An attribute introduces additional parent-child relationships into concepts defined in other languages, without invasively changing the definition of that other language. By making an attribute apply to `BaseConcept`, the super concept of all concepts (comparable to `java.lang.Object`), attributes can be made applicable to arbitrary program elements, effectively supporting AOP-like introductions **TODO**►cite◄ on language level. Attributes are typically used for adding "meta data" such as documentation, presence conditions in product line engineering **TODO**►cite VoelterWorkshopPaper◄ or requirements traces.

MPS comes with a DSL for expressing type system rules. It is based on a unification engine and supports type inference. For typing rules that should be extensible, MPS supports so-called overloaded operation containers. This mechanism supports plugging in additional typing rules by adding them to the language definition. For example, the typing rules for binary operators in C are implemented this way. The language extension that adds types with physical units provides additional typing rules for the case when the left and/or right argument of a binary operator is a type with a physical unit.

Regarding transformations, MPS distinguishes two cases. The first one covers text generation. Text generators are relatively inflexible regarding modularization, but they are used only for the lowest level languages (typically GPLs covering D_0 such as Java or C). All other "generations" happens by transformation rules that work on the projected tree, typically using assimilation. For example, the concepts in a state machine extension to C are transformed back to the concepts in the C language it extends. Overwriting existing generators is also possible by modularizing generators in a suitable way and then using generator priorities to make the overriding generator run before the overridden one. It is also possible to define hooks in generators by generating output elements whose only purpose is to be transformed further by subsequent generators. By plugging in different subsequent generators, the same hook can be transformed into different end results.

Note that MPS does not provide support for systematically handling semantic interactions. Transformations will just fail in case an unintended interaction occurs. To avoid this, transformations have to be designed carefully. Note that a simple approach to lessening the problem would be to be able to declare a language as being *incompatible* with another one. In that case an error could be reported as soon as the languages are used together in a fragment.

SDF/Spoofax **TODO**►EV◄

1.8.7 Cascading

There is an additional **TODO**►◄

1.9 Concrete Syntax

A good choice of concrete syntax is important for DSLs to be accepted by the intended users, particularly if the users are not programmers. There are a couple of major classes for DSL concrete syntax[16]: *Textual* DSLs use traditional ASCII or Unicode text. They basically look and feel like traditional programming languages. *Symbolic* DSLs are textual DSLs with an extended set of symbols, such as fraction bars, mathematical symbols or subscript and superscript. *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

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

The Fortress programming language is close to this.

G. L. S. Jr. Parallel programming and parallel abstractions in fortress. In *IEEEpact*, page 157, 2005 diagrams. *Tables and Matrices* are a powerful way to represent certain kinds of data and can play an important part for DSLs.

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

When to use which form We do not want to make this section a complete discussion between graphical and textual DSLs — a discussion, that is often heavily biased by previous experience, prejudice and tool capabilities. Here are some rules of thumb. Purely textual DSLs integrate very well with existing development infrastructures, making their adoption relatively easy. They are very well suited for detailed descriptions, anything that is algorithmic or generally resembles (traditional) program source code. Symbolic notations can be considered "better textual", and lend themselves to domains that make heavy use of symbols and special notations. Graphical notations are very good for describing coarse-grained structures and for showing relationships between entities. Finally, tables are very useful for collections of similarly structured data items, or for expressing how two independent dimensions of data relate.

Pension Plans: The pension DSL uses mathematical symbols and notations to express insurance mathematics. 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. ◀

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

Embedded C: The primary syntax is textual. However, for state machines, tabular notations are supported as well. The projection can be changed as the program is edited, rendering the same state machine textually or as a table. Graphical notations will be added in the future. ◀

Fountains: The fountain DSL uses graphical visualizations to render diagrams of the hardware structure, as well as a graphical state charts representing the underlying state machine. ◀

Relationship to Hierarchical Domains Domains at low D are most likely best expressed with a textual or symbolic concrete syntax. Obvious examples include programming languages at D_0. Mathematical expressions, which are also very dense and algorithmic, use a symbolic notation. As we progress to higher Ds, the concepts become more and more abstract, and as state machines and block diagrams illustrate, graphical notations become useful. However, these two notations are also a good example of language embedding since both of them require expressions: state machines in guards, and block diagrams as the implementation of blocks. Reusable expression languages should be embedded into the graphical notations. Tool support for the integration of graphical and textual concrete syntaxes, including IDE support, become necessary. In case this is not supported by the tool, viewpoints may be an option. One viewpoint could use a graphical notation to define coarse-grained structures, and a second viewpoint uses a textual notation to provide "implementation details" for the structures defined by the graphical viewpoint[17].

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

Embedded C: The state machine extension of C will be supplanted with a graphical notation later this year. 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. ◄

Selection of concrete syntax is simple for domain user DSLs if there is an established notation in the domain already. The challenge then is to replicate it as closely as possible with the DSL, while cleaning up possible inconsistencies in the notation (since it had not been used formally before). I like to use the term "strongly typed word".[18]

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

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

There are very few DSLs where a purely graphical notation makes sense. In most cases, textual languages are embedded in the diagrams: state machines have expressions embedded the guards and statements in the actions; component diagrams use text for specifications of operations in interfaces, maybe even with expressions for preconditions; block diagrams use a textual syntax for the implementation/parametrization of the blocks. A text box without language support should only be used as a last resort. We suggest using a textual notation, with additional graphical visualizations.

2
High-Level Drivers

In this section we discuss higher-level design forces that are not directly reated to one particular design dimension (those had been discussed in the previous chapter). However, these forces still influence DSL design.

Model Purpose The model purpose defines what the models should be used for. Typical purposes include code generation or interpretation, consistency checking, formal analysis, documentation, and communication among several stakeholders. The DSL will look different for each of these purposes.

> **Example:** If a DSL should be used for model checking and formal analysis, the DSL will probably restrict expressivity. All relevant abstractions will be linguistic abstractions, in-language abstraction will be very limited. ◄

Target Audience DSLs targetted to software developers will be quite different from those targetetted to domain experts. In the former case, integration with existing tools (IDE, CVS, editors) is crucial. In the latter case, a syntax that closely resembles potentially existing domain notations is crucial. The language may also be much more "keyword-heavy" compared to languages oriented to developers, who are used to working with a limited set of orthogonal concepts, using those to build higher level abstractions.

> **Embedded C:** This language is oriented towards developers. It is thus fundamentally C plus a number of extensions. The language is mostly textual. ◄

> **Pension Plans:** This DSL is targetted to domain experts. A mix between textual, symbolic and graphical notations is used. The testing notations is a table, reminding users of Excel. Since before using the DSL no version control system was used, a repository-

based approach was selected for collaboration. ◀

Scoping and Domain Definition Defining the domain, the scope for the DSL is important. A too narrow definition renders the DSL unusable for relevant problems. A too wide definition will make the DSL too generic, and hence less expressive. A domain can be defined bottom-up, by identifying patterns and idioms in existing D_{-1} programs; in this case scoping is simple. If a domain is defined top-down, based on business requirements, scoping is much harder and may require significant analysis effort.

Embedded C: The domains for the base language is C, so this is a trivial corner case. For the extensions, the scope is defined by identifying ◀

Fountains: ◀

Open Or Closed User Community If the set of users are known and accessible for the DSL designers, it will be much easier to evolve the language over time because users can be reached, making them migrate to newer versions. Alternatively, the set of all models can be migrated to a newer version using a script provided by language developers. In case the set of users, and the DSL programs, are not easily accessible, much more effort must be put into keeping backward compatibility, the need for evolution should be reduced. This can be achieved also by using more abstract, composable language constructs as opposed to "a keyword for everything".

Target Audience Developers -> Text Integration into text infrastructures coverage maybe not so important, compensate "Outside" completeness not important, can write low level code collab via checkinout DomEx -> whatever notation realtimer collab coverage, completeness

Extension by Users more imp for developers not very important for DomEx, if coverage 100via lib: good for developers, no "direct" tool support via lang ext: better for DE with annotations, viewpoints Views, Metadata

Integration with Ext Lang

Metadata stored in model -> projection stored in VCS -> integration with those! comments in text only or comments in model

Infrastructure Integration imp for devs better for text (aot projection)

Collaboration checkinout -> partitioning imp for devs file storage gran is partition realtime perhaps more imp for domexp repository gran may be element

Language Size complexity Stakeholders

Model Size Incremental, Impact Analysis

Multiple Stakeholders if so, viewpoints or after the fact views

Ad Hoc vs. Long Lived

Rate of Language Evolution high ->

Domain-specificness, DOmain Size

We look at tooling alternatives, but don't look at how the tools are implemented (e.g. index for performance; pure IDE support: refactoring, CC, etc we consider as given. Also impl strategy: linking, scoping, impact analysis; classes of tools, but not specific tools.)

Embedded C: The model purpose is the generation of efficient C programs for embedded software, for different platforms. A secondary purpose is formal analysis of the programs. The target audience is software developers. We stick to mostly textual notations. The program files are stored in existing VCS. Domain definition is bottom-up: the higher D languages provide abstractions for common C patterns and idioms. The user community at this point is open, since the code is open source. So far we have managed to remain backward compatible. ◀

Component Architecture: The model purpose is the description of the architectural structure of component-based OSGi applications. Code generation to Java and to OSGi manifests is supported. Target audience is developers; models are stored as text files and checked into SVN, the editor is integrated into Eclipse. Domain scoping is top-down, based on the architecture of the system for which the DSL has been built, as well as bottom-up, driven by OSGi. The user community is closed (the developers of an enterprise system), existing models can (and have been) migrated via a script. ◀

Pension Plans: ◀

Fountains: ◀

3
Tooling Influences

3.0.1 Projection vs. Parsing

There are two main approaches for implementing external DSLs. The traditional approach is parser-based. A grammar specifies the sequence of tokens and words that make up a structurally valid program. A parser is generated from this grammar. A parser is a program that recognizes valid programs in their textual form and creates an abstract syntax tree or graph. Analysis tools or generators work with this abstract syntax tree. Users enter programs using the concrete syntax (i.e. character sequences) and programs are also stored in this way. Example tools in this category include Spoofax and Xtext.

Advantages of this approach include simple integration into existing text-based infrastructures, low adoption barrier, and editors working as expected. Disadvantages include a limitation to purely textual notations and limited support for modularization and composition (for some tools).

Projectional editors (also known as structured editors) work without parsers. Editors directly modify the abstract syntax tree. Projection rules then render a textual (or other) representation of the program. Users read and write programs through this projected notation. Programs are stored as abstract syntax trees, usually as XML. As in parser-based systems, backend tools operate on the abstract syntax tree. Projectional editors are well known from graphical editors, virtually all of them are projectional editors. However, they can also be used for textual syntax. While in the past projectional text editors have gotten a bad reputation, as of 2011, the tools have become good enough, and computers have become fast enough to make this approach feasible, productive and convenient to use. Example tools in this category include the Intentional Domain Workbench (`http://intentsoft.com`) and JetBrains MPS.

Advantages of projectional editors include support for unparseable notations, support for non-textual (i.e. graphical, tabular and symbolic) notations in the "text editor", as well as the ability to annotate arbitrary meta data on programs and after the fact views. The disadvantage is that models aren't stored as text, and editing, diffing and merging must be done with the LWB. Editors feel a little bit different, so the adoption barrier is a little bit higher.

3.0.2 Files vs. Repository

Collab Mode Integration Granulatity DB as cache

4
Process

The challenge of developing reusable stuff

4.1 DSLs, Process, Agility

Notice that DSLs are not a methodology, they are a (very powerful) ingredient to a developer's toolkit. Except for managing the supplier-consumer relationship between DSL developers and DSL users, the use of DSLs doesn't have any significant consequences for the development process as a whole. You can still do XP or Scrum. You should still test automatically (we will cover language testing in this book), run a nightly build and so on. And just like when using frameworks or sophisticated technologies, you might want to invest a couple of iterations at the beginning of a project to get a head start with the technology, possibly only delivering limited (obvious) customer value. There is no avoiding the fact that you can only use a DSL after it has been created. But you want to make sure to develop the DSL iteratively and incrementally as you learn about the domain in a project, otherwise you will probably build languages nobody will want or be able to use.

Also, since modern language workbenches are very flexible and support rapid evolution of languages, you will want to make sure that you'll show actual working prototypes of the language (and the tooling) to your future users as you incrementally develop the language. In most cases, there is no need for (technical) mockups - it is quite useful to start and experiment with capturing some of the core knowledge in actual notation on whiteboards or simple text documents, though.

4.2 Discussion

4.2.1 Related Work

Diomidis Spinesslis discusses a set of design patterns for DSLs ([1]). Some of them relate directly to our work: the *Piggypack* pattern describes DSLs that are embedded in a GPL, and the DSL constructs are mapped to GPL constructs for execution. The *Pipeline* pattern addresses the step-wise transformation of higher D programs into lower D programs, which are subsequently translated further. He also defines *Language Extension* to mean that a language has additional features compared to its base language, *Language Specialization* removing concepts from a base language. The rest of his paper looks at common use cases for DSLs and DSL implementation techniques. Our work is different in that it aims at a much more complete coverage of the design dimensions, and provides a conceptual framework (the domain hierarchy and composition) to systematically describe DSLs.

[1] D. Spinellis. Notable design patterns for domain-specific languages. *jss*, 56(1):91–99, 2001

Mernik, Heering and Sloane's paper on How and When to Build a DSL [2] provides a comprehensively broad, but relatively shallow overview over the DSL space. It provides patterns that describe the decision process that leads to building DSLs, then connects to some of the design patterns from Spinellis' paper, and finally delves down into actual implementation approaches. The paper also looks at a couple of supporting tools. The strongest aspect of this work is the huge set of industry examples. Our work is more restricted in that it covers only DSL design and not all the decision process involved in DSL development. We also provide much more detail on the core aspects of language design.

[2] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005

Just like in our paper, Zdun and Mark Strembeck ([3]) generalize from a set of DSLs of different sizes, contexts and target platforms. However, instead of focusing on the design of the languages, they focus on the activities involved in building the DSLs, leading to a tailorable DSL engineering process.

[3] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *SPE*, 39(15):1253–1292, 2009

In their Worst Practices for Domain-Specific Modeling [4], Kelly and Pohjonen discuss recurring problems in the implementation of graphical DSLs. The worst practices are based on actual projects, mostly from Metacase customers. Many of the bad practices are not directly relevant to designing (textual) external DSLs, but help in getting a DSL-based development project set up well: *Analysis Paralysis* refers to "Wanting the language to be theoretically complete, with its implementation assured" — i.e. not enough pragmatism and iteration during the design process. *Lack of Domain Understanding* addresses the problem of "Insufficiently understanding the problem or solution domains". *3GL: Visual Programming* refers to the pitfall of "Duplicating

[4] S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009

the concepts and semantics of traditional programming languages" in the DSL.

Language modularization, composition and extension is an important aspect of DSL design as we describe it in this paper. While building single external DSLs is a pretty well understood and documented topic (as explained in Terence Parr's book [5] and the user guides to the tools mentioned in this paper), modularization, composition and extension of languages and IDEs are still an area of active research. For example, KHEPERA [6] is a system for progressively translating DSLs down along the D hierarchy. OpenJava [7] and the Java Syntactic Extender [8] are macro systems for Java that support the incremental extension of Java with additional syntax. The extensions are translated back to the Java idioms from which they abstract. In [9], Bravenboer, Dolstra and Visser discuss an approach for embedding of arbitrary guest languages into arbitrary host languages. Xoc [10] and the Modular Embedded DSL [11] provide similar functionality for C. MontiCore [12] supports the development of modular languages, where binary composability, so no languge sources such as grammars are required for composition. Finally, Hemel and Visser describe PIL [13], an intermediate language intended as to be used as a reusable transformation backend for DSLs - DSLs are transformed to PIL, reusing a set of existing reusable PIL backend for several target languages.

[5] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2010

[6] R. E. Faith, L. S. Nyland, and J. Prins. Khepera: A system for rapid implementation of domain specific languages. In *DSL*, 1997

[7] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *oorase*, pages 117–133, 1999

[8] J. Bachrach and K. Playford. The java syntactic extender. In *OOPSLA*, pages 31–42, 2001

[9] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *GPCE*, pages 3–12, 2007

Acknowledgements We thank Nora Ludewig and Andreas Graf for their feedback on previous versions of the paper.

finitive ANTLR Reference: Building Domain-Specifi
c Languages. Pragmatic Programmers, May 2007.