Modularity and Composition ~~for (~~Domain-Specific~~)~~ Languages

Eelco Visser
Delft University of Technology
visser@acm.org

Markus Voelter
independent/itemis
voelter@acm.org

Abstract

Domain-Specific Languages (DSLs) are languages with high expressivity for a specific, narrow problem domain. They are a powerful tool for software engineering, because they can be tailor-made for a specific class of problems. However, because of the large degree of freedom in designing DSLs, and because they are supposed to cover the right domain, completely, and at the right abstraction level, DSL design is also hard. In this paper we present a framework for describing and characterizing external domain specific languages. We identify eight design dimensions that span the space within which DSLs are designed: expressivity, coverage, semantics, separation of concerns, completeness, large-scale model structure, language modularization and syntax. We illustrate the design alternatives along each of these dimensions with examples from five different case studies. These have been selected for their diversity in context, style and implementation technologies. The paper concludes with an outlook on further steps towards comprehensive DSL design guidance.

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. $T(p_{L_1}) = p_{L_2}$.

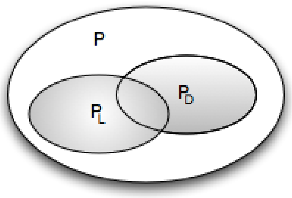

Figure 1. Programs, Languages, Domains

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 pro-

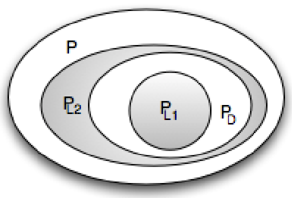

Figure 2. Languages L1 and L2 under-approximate and over-approximate domain D.

grams 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.

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 **??**). 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 **??**).

Programs as Trees of Elements We assume models to be trees of elements. We define E as a set of program elements. Such a tree is created either via parsing, or directly, when projectional editing is used.

A program may be composed from several *program fragments*, or *fragments* for short. A fragment that has references to elements outside itself is called a *dependent* fragment. It cannot be completely understood or processed without the transitive closure of all other fragments it depends on.

A Fragment f is a standalone tree, i.e. there is a root model element that has no parent. E_f is the set of program elements in a fragment.

A language l contains a set of language concepts C_l. In a fragment, each program element e is an instance of a concept c of some language. We define the concept-of relation co to map a set of elements to the set of corresponding concepts $co \Rightarrow E \rightarrow C$ so that $\forall e \in E \mid co(e) \in C$.

Similarly we define the language-of relation lo to map a set of concepts to the set of languages that contain these concepts: $lo \Rightarrow C \rightarrow L$.

Finally, we define a fragment-of relation fo that maps a set of program elements to the set of fragments that contain these elements: $fo \Rightarrow E \rightarrow F$.

Note that for convenience, we also use these relations for single elements, as in $lo(co(e) = l$ where e is a program element and l is a language.

We also define a number of relations between program elements. Ref_f is the set of non-containing cross-references between program elements in a fragment f. Each reference r in Ref_f has the properties $from$ and to, that refer to the two ends of the reference relationship. Cdn_f is similar, but it denotes the set of containment relationships. Each $c \in C$ has the properties $parent$ and $child$. 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 $i \in Inh$ has the properties $super$ and sub.

A core concept in language modularization and composition is the notion of independence. An *independent language* does not have any dependencies on other languages. An independent language l can be defined as a language for which the following hold:

$$\forall r \in Ref_L \mid lo(r.to) = lo(r.from) = l \qquad (1)$$

$$\forall s \in Inh_L \mid lo(s.super) = lo(s.sub) = l \qquad (2)$$

$$\forall c \in Cdn_L \mid lo(c.parent) = lo(c.child) = l \qquad (3)$$

An *independent fragment* is one that contains program elements that are instances of concepts in a single, independent language l:

$$\forall e \in E_f \mid lo(co(e)) = l \qquad (4)$$

and all references between elements in the fragment have to point to another element in that same fragment, so

$$\forall r \in Ref_f \mid fo(r.from) = fo(r.to) = f \qquad (5)$$

We can now discuss the various different relationships of fragments and languages.

2. Classification

2.1 Langauage Referencing

2.1.1 Thumbnail

Language Referencing (Fig. 3) covers the case where program elements from two independent fragments expressed with two different languages reference each other. The referencing language l_2 depends on the referenced language l_1 because at least one concept in the l_2 references a concept from l_1.

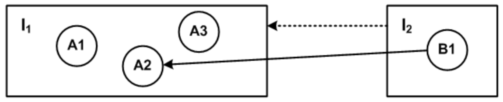

Figure 3.

2.1.2 Formal Definition

We call language *referencing* the case where a fragment f_2 depends on f_1, and f_1 and f_2 are expressed with different languages l_1 and l_2. In case of f_2 depending on f_1, l_2 cannot be an independent language anymore, since it references elements from l_1. While (2) and (3) continue to hold, (1) does not. Instead

$$\forall r \in Ref_{l_2} \mid lo(r.from) = l_2 \ \wedge \ lo(r.to) = (l_1 \vee l_2) \quad (6)$$

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

2.1.3 Uses Cases and Examples

Viewpoints A domain D can be composed from different concerns. Fig. 4 shows $D_{1.1}$ composed from the concerns A, B and C. For embedded software, these could be component and interface definitions (A), component instantiation and connections (B), as well as scheduling and bus allocation (C). To describe a complete program for D, the program needs to address all the concerns.

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. 5 illustrates the two different approaches.

As an example, consider a DSL for refrigerator configuration. The DSL consists of three parts. The first part describes the hardware structure of refrigerators installations. The structure is defined by composing various hardware features by inheritance into complete appliances. The various features contribute actual model elements that represent hardware elements. The second part describes the cooling al-

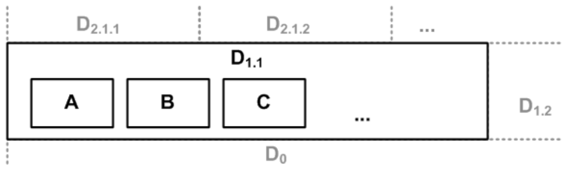

Figure 4. A domain may consist of several concerns. A domain is covered either by a DSL that addresses all of these concerns, or by a set of related, concern-specific DSLs.

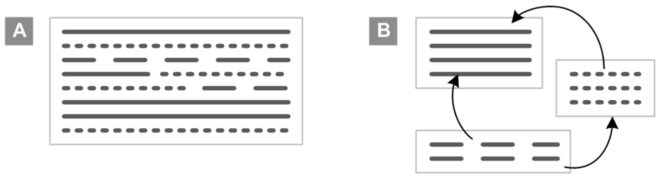

Figure 5. Part A shows an integrated DSL, where the various concerns (represented by different line styles) are covered by a single integrated language (and consequently, one model). Part B shows several viewpoint languages (and model fragments), each covering a single concern. Arrows in Part B highlight dependencies between the viewpoints.

gorithm using a state-based, asynchronous language. Cooling programs refer to hardware features and they can access the properties of hardware elements from expressions and commands. The second aspect provided by the DSL is testing. A cooling test can test and simulate cooling programs.

Each of these concerns are implemented as a separate language with references between them. TODO▶Elaborate◀

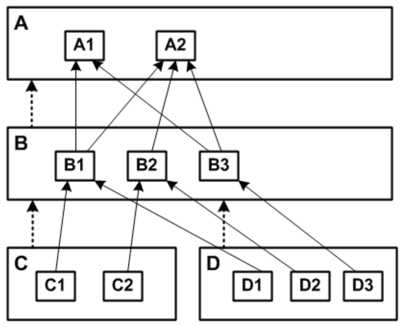

Figure 6. Progressive refinement: the boxes represent models expressed with corresponding languages. The dotted arrows express dependencies, whereas the solid arrows represent references between model elements.

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. 6). The references between model elements are called traces[**?**]. 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 [**? ?**]).

2.1.4 Implementation Challenges

Syntax No deep syntax composition is required since the referencing language only points to concepts defined in another language. Only references have to be supported in the referencing language. In the vast majority of cases such a reference is simply a (qualified) name. The name resolution phase creates the actual cross-reference.

Type System

Transformations

IDE If viewpoints are used, the concern-specific languages, and consequently the viewpoint models, should have well-defined dependencies; cycles should be avoided. 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"). 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.

TODO▶Table that compares the four approaches reg crit above◀

2.1.5 Solution Approaches

2.2 Language Extension

2.2.1 Thumbnail

2.2.2 Formal Definition

A language l_2 extends l_1 if it adds additional language concepts, some of them being subconcepts of concepts defined in l_1, so they can be used inline. Consequently, while l_1 remains independent, l_2 becomes dependent on l_1 since

$$\exists i \in Inh(l_2) \mid i.sub = l_2 \ \wedge \ i.super = l_1 \qquad (7)$$

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

$$\forall e \in E_f \mid lo(e) = (l_1 \vee l_2) \qquad (8)$$

Another way of saying this is that $C_f \subset (C_{l_1} \cup C_{l_2})$. We call such a fragment *heterogeneous*, and those where $C_f \subset C_l$ we call *homogeneous*. Note also that for hereogeneous fragements (3) does not hold anymore, since

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = \\ (l_1 \vee l_2) \ \wedge \ lo(co(c.child)) = (l_1 \vee l_2) \qquad (9)$$

2.2.3 Uses Cases and Examples

Domain Hierarchy The subsetting of domains naturally gives rise to a hierarchy of domains (Fig. 7). 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

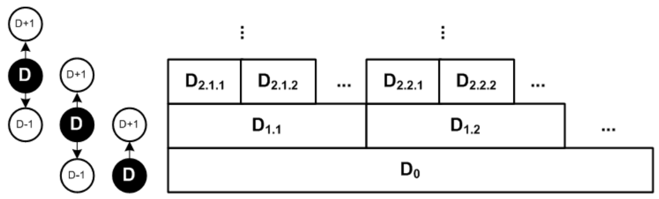

Figure 7. Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

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.

Progressive language extension fits well with the hierarchical domains introduced above: a language L_B for a domain D may extend a language L_A for D_{-1}. Since L_B also contains all of L_A, it has 100% coverage of D_{-1} and it is also a complete language, since D_{-1} concepts can be used whenever necessary. Nonetheless it contains concepts specifically for the domain D, making analysis and transformation of those concepts possible without pattern matching and semantics recovery. As explained before, the new concepts are reified from the idioms and patterns used in the L_{D-1} language, when used for D. Language semantics are typically defined by mapping the new abstractions to just these idioms (see Section **??**) *inline* (a process also called *assimilation*. Assimilation transforms a heterogeneous fragment (expressed in L_D and L_{D+1}) into a homogeneous fragment expressed only with L_D. The idea of progressively extending languages is related to Guy Steele's concept of growing a language [**?**]. Language extension is especially interesting if D_0 languages are extended, making a DSL an extension of a general purpose language.

2.2.4 Implementation Challenges

Syntax

Type System

Transformations

IDE

2.2.5 Solution Approaches

2.3 Language Reuse

2.3.1 Thumbnail

2.3.2 Formal Definition

Given are two independent languages l_1 and l_2 and two fragment f_1 and f_2. f_2 depends on f_1, so that

$$\exists r \in Ref_{f_2} \mid fo(r.from) = f_2 \ \wedge \ fo(r.to) = (f_1 \vee f_2) \quad (10)$$

Since l_2 is independent, it cannot directly reference concepts in l_1. This makes l_2 reusable with different languages (this is in contrast to language referencing, where concepts in l_2 reference concepts in l_1).

One way of making dependent fragments possible while retaining two independent languages is using some kind of adapter language l_A where l_A *extends* l_2 and

$$\exists r \in Ref_{l_A} \mid lo(r.from) = l_A \ \wedge \ lo(r.to) = l_1 \quad (11)$$

2.3.3 Uses Cases and Examples

2.3.4 Implementation Challenges

Syntax

Type System

Transformations

IDE

2.3.5 Solution Approaches

2.4 Language Embedding

2.4.1 Thumbnail

2.4.2 Formal Definition

Embedding is similar to reuse in that there are two independent languages l_1 and l_2. However, instead of just establishing references between two fragments, we now embed instances of concepts from l_2 in instances of concepts from l_1. We create a heterogeneous fragment f where

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge \\ lo(co(c.child)) = (l_1 \vee l_2) \quad (12)$$

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. This apparent dichotomoy can be solved by using an adapter language l_a that extends l_1, which

$$\exists c \in Cdn_{l_A} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \quad (13)$$

2.4.3 Uses Cases and Examples

2.4.4 Implementation Challenges

Syntax

Type System

Transformations

IDE

2.4.5 Solution Approaches