

Modularity, Reuse and Composition for Languages

Eelco Visser

Delft University of Technology
visser@acm.org

Markus Voelter

independent/itemis
voelter@acm.org



Abstract

This paper presents a classification of approaches for language modularization, reuse and composition. We identified four different approaches: referencing, extension, reuse and embedding. They differ regarding the dependencies between the participating languages and whether they require syntactic composition or not. For each of the four approaches, the paper presents a formal definition, usage scenarios and examples, implementation challenges regarding syntax, type systems, transformations and IDE support, as well as an overview over how three different DSL tools (SDF/Spoofax, MPS and Xtext) support these approaches.

1. Introduction and Motivation

Traditionally, programmers have used general purpose languages (GPLs). The name general-purpose stems from the fact that they can be used for any programming task — they are turing complete, and provide means to build abstractions using classes, (higher-order) functions or logic predicates, depending on the particular GPL. Traditionally, a complete software system has been implemented using a single GPL, plus a couple of configuration files. More recently, however, this has been changing. Systems are built using a multitude of languages. This change comes from two main reasons.

One reason is a consequence of more and more powerful and complex infrastructures. For example, web applications consist of business logic on the server, a database backend, business logic on the client and presentation code on the client. A particular language stack could use Java, SQL, JavaScript and HTML for those aspects. A complete system is implemented by using these languages together.

The second reason that motivates the move away from using a single GPL to implement a software system is domain-specific languages (DSLs). These are specialized, often small languages that are optimized for expressing pro-

grams in a particular application domain. Such an application domain may be a technical domain (in which case SQL, mentioned in the previous paragraph, could be considered a DSL) or an actual business domain, such insurance contracts, refrigerator cooling algorithms, or state-based programs in embedded systems. DSLs support these domains more effectively than GPLs because they provide linguistic abstractions for common idioms in those particular domains. Using custom linguistic abstractions makes the code more concise and more easily accessible to formal analysis, verification, transformation and optimization.

So if we use a multitude of languages to implement software systems, then the question is, how these languages are integrated regarding syntax, semantics and IDE support. Traditionally, syntactic integration has been very hard and hence is often not supported for a particular combination of languages. So programs written in different languages reside in different files. References between "common" things in these different programs are implemented by using agreed-upon identifiers that are must be used consistently in the different partial programs expressed in different languages. For some combinations of languages, the IDE may be aware of the "integration by name" and check for consistency. In some rare cases, syntactic integrations between specific pairs of languages has been built, for example, embedded SQL in Java.

However, building specialized integrations between two languages is very expensive, especially if IDE support is to be provided as well. So, this is done only for combinations very widely used languages, if at all. Building such an integration between for example Java and a company-wide DSL for financial calculations is out of the question based on this approach. Embedding, say, SQL code into Java simply as strings is also not a satisfying option, because no static checking, code completion and static optimization are available. A more systematic approach for language modularization, reuse and composition is required.

In this paper, we introduce a systematic approach for characterizing modularized languages. We identify four different approaches, which we introduce in Section 3: language referencing, extension, reuse and embedding. For each of these approaches we provide a concise thumbnail and a formal definition, usage scenarios and an example.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Section ??, describes challenges that need to be addressed when implementing the particular composition approach, and Section ?? describes how Xtext, SDF/Spoofax and MPS address these challenges. We conclude the paper with related work (Section 6) and a Summary. In the rest of this section we outline the general challenges in language modularization and briefly introduce the tools we use as examples. To provide foundation for our discussion, the next section Section 2 introduces terminology and concepts.

1.1 Challenges in language modularization

We have identified the following four broad classes of challenges when implementing language modularity and composition. For each of our modularization approaches, we provide details on these.

Syntax Many language description formalisms, such as $ll(k)$, are not closed under composition. Combining two valid grammars may result in an invalid, ambiguous grammar. So, when languages should be combined, the languages have to be described with formalisms that support combinations of independently developed grammars.

Type System The languages used for describing type systems and constraints have to support enhancement or combination with other type system definitions.

Transformations Existing, reusable language may already come with transformations for certain target platforms. These may have to be reused, combined or extended.

1.2 Example Tools

In this paper we use three different tools to illustrate if and how our language modularization approaches are implemented. We have chosen Xtext, SDF/Spoofax and MPS because they work quite differently and support different aspects of our approaches in different ways. All of them are language workbenches, which, from a language definition, derive all the ingredients necessary to edit and process programs written in the particular language. Specifically, they all derive an IDE that supports code completion, syntax highlighting and static error checking.

We consider IDE support an important ingredient for effectively working with languages. While not all of these tools support all of the composition approaches discussed in this paper, they all provide IDE support for all the composition approaches they do support. So if, for example, a tool supports referencing elements from one program in another one, then the IDE will support cross-language go-to-definition and find references. If a tool supports embedding one language in another one, then the tool supports code completion, static error reporting for the combined languages.

The following paragraphs briefly describe the tools and the reasons for choosing them in this paper. All of them are Open Source and hence available for experimentation.

Xtext As part of the Eclipse Modeling framework, Xtext is based on EMF. It supports the definition of textual concrete syntax for Ecore-based meta models. Xtext supports many advanced IDE aspects. As we will see, its support for language modularization is limited, partly because of its reliance on the antlr parser, which is $ll(k)$. The main reason for including in this discussion is that it is widely used - it can be considered that industry standard tool for the definition of textual DSLs and IDEs.

SDF/Spoofax SDF is a syntax definition formalism based on GLR parsing, which, consequently supports language modularization and composition quite nicely. Spoofax is an Eclipse-based IDE framework that uses SDF internally. It is developed by Eelco Visser's SERG group at the TU Delft. While it is not yet widely used, a couple of non-trivial languages and IDEs have been built with it (mobl, webdsl). We include it in the discussion here mainly because of its extensive support for modularization and composition.

MPS MPS is developed by JetBrains. It is fundamentally different from the two other tools because it uses projectional editing. No parser is involved, changes to a program's textual representation directly change the underlying program tree. As a consequence, many of the challenges of grammar composition don't exist in the first place. In spite of the fact that MPS isn't used widely yet, we have included in this discussion because of its very good support for modularization and composition, and its use of projectional editing. **TODO**►Refer to GTTSE Paper here. ◀

2. Concepts and Terminology

2.1 Programs, Languages, Domains

In this paper we will be talking a lot about *programs*, *languages*, and *domains*, so we explain these terms here.

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

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.


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 . 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 . General purpose languages are those that can express all programs P . Consequently, they can also express P_D for any D , but they may not be particularly efficient (regarding program size or complexity) in doing so. This is where DSLs have an advantage.


As a consequence of DSLs being specialized languages for expressing some limited domain D naturally leads to the need for combining multiple DSLs that cover the various domains relevant for fully implementing a particular system. To reduce the overhead of creating all those DSLs, it is desirable to reuse, adapt and extend existing languages. This, in turn, results in the need for language modularization.

2.2 Programs as Trees of Elements

TODO►Why are trees important (containment stuff) **◀** We assume programs 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. In addition to the primary containment hierarchy, the programs also have non-containing cross-references.


A program  be composed from several *program fragments*, or *fragments* for short. A Fragment f is a **standalone** tree, i.e. **there is** a root program 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 . We use the term concept to refer to concrete syntax, abstract syntax plus the associated type system rules and constraints. In a fragment, each program element e is an instance of a concept c of some language. 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 that represent 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*. $Refs_f$ is the set of non-containing cross-references between program elements in a fragment f . Each reference r in $Refs_f$ has the properties *from* and *to*, which refer to the two ends of the reference relationship. **TODO►**Do we need those two also for language concepts, or do we always use them with fragments? **◀**

TODO►Is a concept unique to one language? **◀** **TODO►**Is a language just a bunch of concepts? **◀**


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: 

$$\forall r \in Refs_l \mid lo(r.to) = lo(r.from) = l \quad (1)$$

$$\forall s \in Inh_l \mid lo(s.super) = lo(s.sub) = l \quad (2)$$

$$\forall c \in Cdn_l \mid lo(c.parent) = lo(c.child) = l \quad (3)$$

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

$$\forall r \in Refs_f \mid fo(r.to) = fo(r.from) = f \quad (4)$$

$$\forall e \in E_f \mid lo(co(e)) = l \quad (5)$$

We also distinguish *homogeneous* and *heterogeneous* fragments. A homogeneous fragment is one where all elements

are expressed with the same language:

$$\forall e \in E_f \mid lo(e) = l \quad (6)$$

$$\forall c \in Cdn_f \mid lo(c.parent) = lo(c.child) = l \quad (7)$$

3. Classification

We now have all the ingredients to discuss the various approaches to language modularization and composition. We have identified the following four approaches: referencing, extension, reuse and embedding. **TODO**►D also want to add restriction (removing stuff) and MPS' annotations?◀

We distinguish the four approaches regarding fragment structure and language dependencies, as illustrated in Fig. 1. Fig. 2 shows the relationships between fragments and languages in these cases.

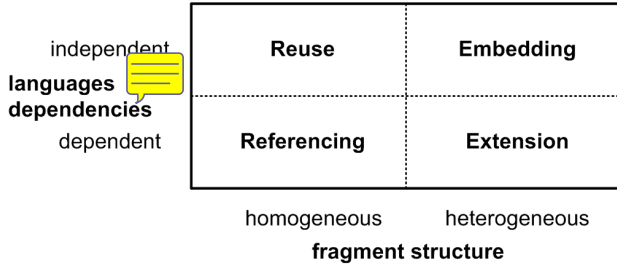


Figure 1. We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies.

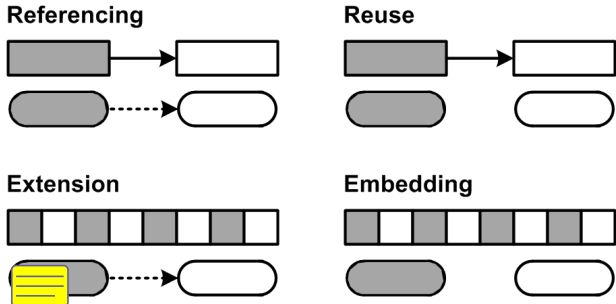


Figure 2. The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

3.1 Language Referencing

TODO►Should be renamed to Combination, I guess.◀

Language referencing (Fig. 6) enables *homogeneous* fragments with cross-references among them, with *dependent* languages. 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 .

A fragment f_2 depends on f_1 , with f_2 and f_1 being 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

$$\forall r \in Refs_{l_2} \mid lo(r.from) = l_2 \wedge lo(r.to) = (l_1 \vee l_2) \quad (8)$$

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

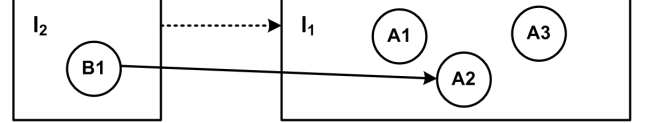



Figure 3. Referencing: Language l_2 depends on l_1 , because concepts in l_2 reference concepts in l_1 . (A note on the notation: we use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

3.1.1 Cases and Examples


Viewpoints A domain D can be composed from different concerns. 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 fragment. 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 concern-specific fragments, that relate to each other in a well-defined way using language referencing. Fig. 4 illustrates the two different approaches. Using separate fragments has the advantage that different stakeholders can modify "their" concern independently of others. It also allows reuse of independent fragments and languages with different referencing languages.

As an example, consider the domain of refrigerator configuration. ~~It consists DSL~~ consists of four concerns. The first concern H describes the hardware structure of refrigerators appliances. The structure is defined by composing various hardware features by inheritance into complete appliances. The second concern A describes the cooling algorithm 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 third concern is testing T . A cooling test can test and simulate cooling programs. Finally, the fourth concern P is parametrization. It is used to configure a specific hardware and algorithm description 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. T is separate from A because tests are not strictly part of the product definition and are enhanced also after a product has been released. Finally, P is separate, because the parameters have to be changed by technicians in the field. Also several parametrizations for the  algorithm exist.

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 levels” of the system (Fig. 5). The references between program elements are called traces[?].

3.2 Language Extension

Language extension Fig. 6 enables *heterogeneous* fragments with *dependent* languages. A language l_2 extending l_1 adds additional language concepts to those of l_1 . To allow the new concepts to be used in the context provided by l_1 , some of them are subconcepts of concepts in l_1 . So, while  remains independent, l_2 becomes dependent on l_1 since

$$\exists i \in \text{Inh}(l_2) \mid i.\text{sub} = l_2 \wedge i.\text{super} = l_1 \quad (9)$$

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

$$\forall e \in E_f \mid lo(e) = (l_1 \vee l_2) \quad (10)$$

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 $C_f \subset C_{l_1}$ we call *homogeneous*. Note also that for heterogeneous fragments (3) does not hold anymore, since

$$\begin{aligned} \forall c \in \text{Cdn}_f \mid lo(\text{co}(c.\text{parent})) &= (l_1 \vee l_2) \wedge \\ lo(\text{co}(c.\text{child})) &= (l_1 \vee l_2) \end{aligned} \quad (11)$$

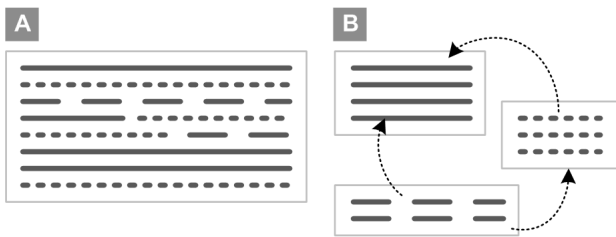


Figure 4. Part A shows an integrated DSL, where the various concerns (represented by different line styles) are covered by a single integrated language (and consequently, one fragment). Part B shows several viewpoint languages (and program fragments), each covering a single concern. Arrows in Part B highlight dependencies between the viewpoints.

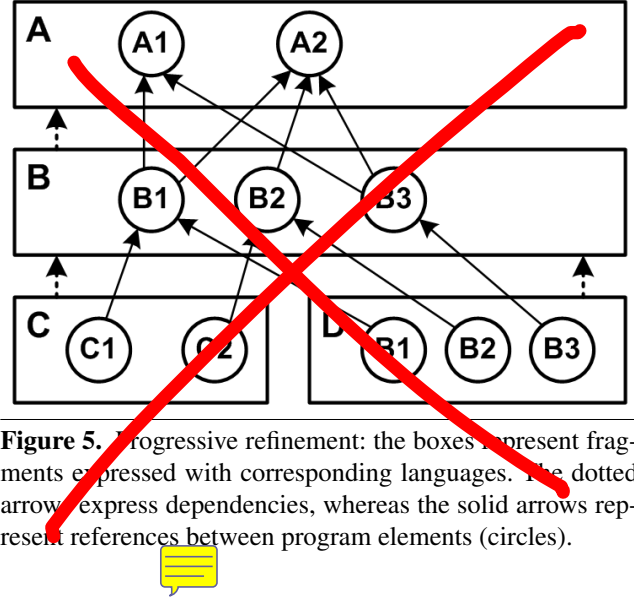


Figure 5. Progressive refinement: the boxes represent fragments expressed with corresponding languages. The dotted arrow express dependencies, whereas the solid arrows represent references between program elements (circles).

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 C from Perl or C from Java) is not language extension; rather it is a form of language referencing. The fragments remain homogeneous.

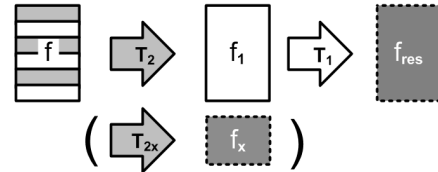


Figure 6. Extension: l_2 extends l_1 . It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1 . As a consequence, l_2 depends on l_2 .

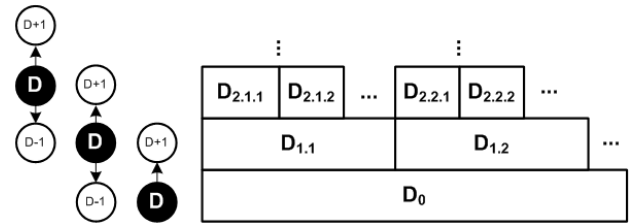


Figure 7. Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

3.2.1 Uses Cases and Examples

Domain Hierarchy Domains are naturally organized in hierarchies (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 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-infinity 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 .

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} . It contains concepts specifically for the domain 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 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.

As an example consider embedded programming. The domain of embedded systems can be considered a subset of D_0 where concepts such as state machines, tasks or data types with physical units are used. On top of the domain of embedded systems, more specialized domains can be cascaded, for example the domain of real-time systems or of safety-critical systems. On top of those, one could imagine the domain of avionics or mission software for spacecraft.

The C programming language is typically used as the GPL in this case. However, there are many useful extensions that could be added to C by means of language extension to address the progressively more specialized domains. State machines, tasks and data types with physical units could be represented as separate extensions of C to address embedded systems in general. Language extensions for real-time systems may include ways of specifying deterministic scheduling and worst-case execution time analysis. For avionics, direct language support for remote communication using some of the bus systems used in avionics could be added.

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 dynamically allocated memory. The extension for safety-critical systems may remove 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 may be added using constraints that report errors in case the restricted language constructs are used in a fragment. Second, often some kind of marker concept (e.g. "safe" modules) are added to the base language. The restriction rules are then enforced for children of these marker concepts (e.g. in a safe module, you cannot use void pointers and downcasts).

3.3 Language Reuse

Language reuse (Fig. 8) 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

$$\begin{aligned} \exists r \in Refs_{f_2} \mid fo(r.from) = f_2 \wedge \\ fo(r.to) = (f_1 \vee f_2) \end{aligned} \quad (12)$$

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 an adapter language l_A where l_A extends l_2 and

$$\exists r \in Refs_{l_A} \mid lo(r.from) = l_A \wedge lo(r.to) = l_1 \quad (13)$$

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 is necessary.

3.3.1 Uses Cases and Examples

Language referencing ~~support~~ reuse of the referenced language. Language reuse, in addition, 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, layouts, and disable/enable strategies. It also supports data binding, which means that data structures are associated with the UI widgets, to enable two-way synchronization between the UI and the data structures. Using language reuse, the same UI language can be used with different data description languages.

Language referencing is not enough because the UI language would have a direct dependency on a particular data

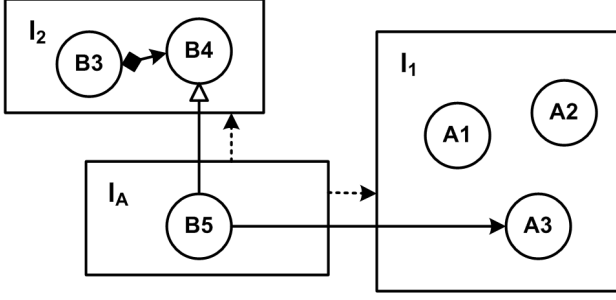


Figure 8. Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1 . To enable this, an adapter language l_A is added. It depends on both l_1 and l_2 , and uses inheritance and referencing to adapt l_1 to l_2 .

description language. Changing the dependency direction to $data \rightarrow ui$ doesn't solve the problem either, because this would go against the generally accepted idiom that UI has dependencies to the data, but not vice versa (MVC pattern).

However, it is generally the case that the referencing language has been built with the knowledge that it will be reused with other languages. The UI language may not know where it gets the data to edit from, but it knows that it will may be combined with another language that provides the data.

3.4 Language Embedding

Language embedding (Fig. 9) enables *heterogenous* fragments with *independent* languages. So it is similar to reuse in that there are two independent languages l_1 and l_2 . However, instead of just establishing references between two homogeneous fragments, we now embed instances of concepts from l_2 in instances of concepts from l_1 . We create a heterogeneous fragment f where

$$\begin{aligned} \forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge \\ lo(co(c.child)) = (l_1 \vee l_2) \end{aligned} \quad (14)$$

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

$$\exists c \in Cdn_{l_a} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \quad (15)$$

3.4.1 Uses Cases and Examples

The best example for language embedding is an embeddable expression language. Expressions are used in many contexts beyond "normal" programming. Examples include guard conditions in state machines, disable/enable conditions in UI specifications or definition of derived fields in data definition languages. Many of the literals (numbers,

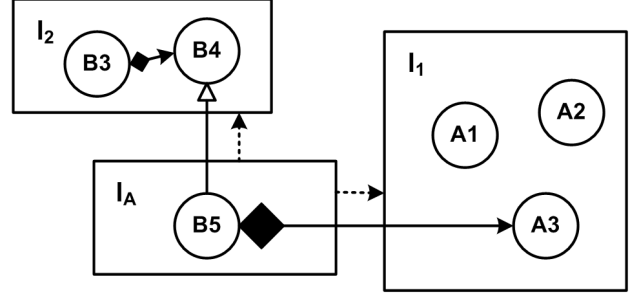


Figure 9. Embedding: l_1 and l_2 are independent languages. However, we still want to use them in the same fragment. To enable this, an adapter language l_A is added. It depends on both l_1 and l_2 , and uses inheritance and composition to adapt l_1 to l_2 (this is the almost the same structure as in the case of reuse; the difference is that $B5$ now contains $A3$, instead of just referencing it.)

strings) and operators will be reusable in all these contexts. Extension is not enough, since the expression language must be embeddable in several, unrelated host languages. Reuse is not enough, the expressions are closely related to their context (state transition, UI widget, data structure), so putting them into separate fragments and referencing them will not do.

Note that, 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 to event parameters, widget content and the values of other data structure fields. These additional expressions will typically reside in the adapter language as well.

TODO►talk about MPS' annotations as well.◀

4. Implementation Challenges and Solutions

4.1 Syntax

Referencing and Reuse In these approaches, fragments remain homogeneous, so "mixing" of concrete syntax is not required. The referencing or reusing language only points to concepts defined in another language. In the vast majority of cases such a reference is simply a (qualified) name and does not have its own internal structure for which a grammar would be required. The name resolution phase creates the actual cross-reference. In the refrigerator example, the algorithm language contains cross-references into the hardware language. Those references are simple, dotted names. In the UI example, the adapter language simply introduces dotted names to refer to fields of data structures.

All three of the tools, Xtext, MPS and Spoofox, support this approach **TODO**►◀

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, the state

machine language embeds state machines into 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. In the embeddable expression language example, the expressions must be integrated with arbitrary host languages.

4.2 Type Systems

Referencing The type system rules and constraints of the referencing language possibly have to take into account the referenced language. Since the referencing language is developed with knowledge about the referenced 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 (a compressor can be turned on or off).

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 language extension defines new types, then those typing rules may have to be overridden to allow the use of existing operators with new types. In the embedded systems example, we have added a language that 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.)

Reuse and Embedding The typing rules that affect the interplay between the two languages can 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 performed in the adapter language.

4.3 Transformation

In this section we use the terms *transformation* and *generation* interchangeably. In general, the term transformation is typically used if one tree of program elements is mapped to another tree, while generation covers the case of creating **test** from program trees. However, for the discussions in this section, this distinction is generally not relevant. In the one place where we do make the distinction, we have made it explicit.

Referencing There are three cases regarding transformation. The first one (Fig. 10 A) propagates the referencing

structure to the two independent target fragments. We call these two transformations single-sourced, since each of them only uses a single, homogeneous fragment and creates a single, homogeneous fragment, perhaps with references among the two. 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.

The refrigerator example uses this case. From the hardware description we an XML file that configures a framework that actually collects the data 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.

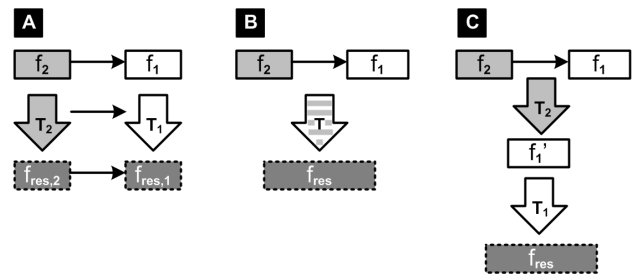


Figure 10. Three alternatives of handling the transformations for language referencing: (A) two separate, dependent, single-source transformation, (B) a single multi-sourced transformation and (C) a preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment, then reusing existing transformations for the referenced fragment (Notation for this and the following illustration: boxes are fragments, fat arrows are transformations, thin arrows are dependencies and shading represents languages.)

The second case (Fig. 10 B) is a multi-sourced transformation that creates one single homogeneous fragment from the two input fragments. This is especially typical 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. 10 C), 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.

Extension As we have discussed above, language extensions are usually created by creating linguistic abstractions

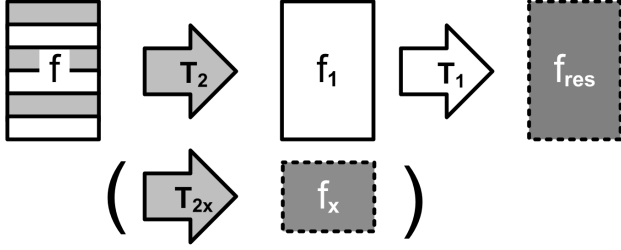


Figure 11. Transformation for language extensions usually happens by assimilation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often some configuration files.

for common idioms of a domain D if it is expressed in a language L_{D-1} . A generator for the new language concepts thus can simply recreated those idioms when mapping L_D to L_{D-1} . 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. 11). In some cases additional files are generated, often configuration files. In the embedded systems state machines example, the state machines are generated down to function that contains a big switch case statement, as well as enums for states and events.

A related case for transformations in language extensions is that the transformation of language concepts from the base language may have to be overwritten. 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, we transform this into a call to a special add function instead of using the regular plus operator. A similar thing happens for assignments. A special function is used to wrap the assignment to perform overflow checking.

TODO►Semantic Interactions!◄

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) case where several extensions to Java, when transformed to pure Java, all want to change the base class of the class in which the extension is embedded. This won't work because Java can only have one base class per 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. **TODO►Not yet really satisfying◄**

Reuse In the reuse scenario, it is likely that both the reused and the reusing language already come with their own gener-

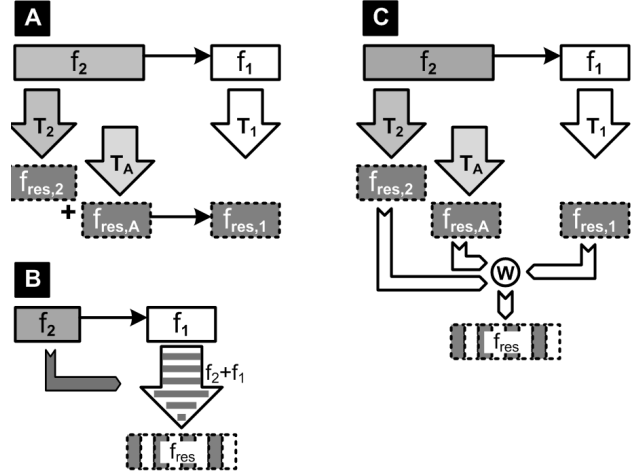


Figure 12. Transformation in the case of reuse comes in three flavors. reuse of existing transformations for both fragments plus generation of adapter code (A), composing transformations (B) or generating separate artifacts plus a weaving specification (C).

ators. If these generators transform to different, incompatible target languages, no reuse is possible. If they transform to a common target languages (such as Java or C) then the potential for reusing previously existing transformations exists.

There are three cases to consider. The first one, illustrated in Fig. 12 A, describes the case where there is an existing transformation for the referenced fragment and an existing transformation for the referencing 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 (example: super class, subclass). In the second case, Fig. 12 B, the existing generator for the referenced fragment has to be changed, or enhanced, with transformation code specific to the referencing language. In this case, some kind of transformation composition mechanism is needed. The third case, Fig. 12 B leaves composition to the target languages. We generate three different independent, homogeneous fragments, and a weaver composes them into one final, heterogeneous artifact. Often, the weaving specification is the intermediate result generated from the adapter language.

Embedding An embeddable language may not come with any reusable generator, since, at the time of writing the embeddable language, one cannot know what to generate from the embeddable language. In that case, when embedding the language, one has to develop a suitable generator. It will typically transform the embedded language elements to the same target language that is also generated to by the embedding language.

If the embeddable language comes with a generator that transforms to the same target language as the embedding lan-

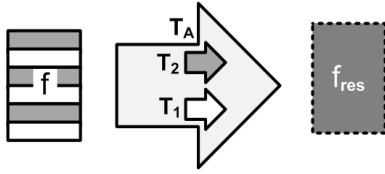



Figure 13. In transforming embedded languages, a new transformation has to be written if the reusable language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the embedded languages.

guage, then the generator for the adapter language can coordinate the two, and make sure a single, consistent fragment is generated. Fig. 13 illustrates this case.

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

5. Tool Support

5.1 Eclipse Xtext

Syntax Language referencing works **nicely** in Xtext. The grammar language comes with special support to reference concepts defined in other languages (or Ecore-based  model in general). Language extension is also **supported**. However, it is possible to extend from at most one base language. Language reuse is **possible** as well. However language embedding is not supported because, of the limitation to extend only one language. This limitation seriously limits language modularization and composition with Xtext.

Constraints/Type Systems Constraints are implemented in Xtext as Java methods that query the AST and report errors if something is wrong. The methods are associated with the particular language concept whose correctness they check. Xtext does not directly support a concise definition of type systems, but third party solutions exist. Xtext also comes with XBase, a reusable expression language that can be "embedded" into languages built with Xtext by having that language extend XBase. 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 embeds/extends XBase.

Transformation Xtext comes with a transformation and code generation language called Xtend. Transformation can be achieved by using a special syntax for object construction. Text generation is supported with special strings that contain a template syntax similar to the one supported by the old Xpand. Xtend supports dependency injection as a first class language concept. If a generator delegates the generation of some part or aspect of the system to an Xtend class that is instantiated via dependency injection, then subclasses, con-

taining different code generation rules, can be injected after the fact. In effect this results in the ability to override parts of generators selectively, if the original generator developer has built in the ability for extension by delegating the relevant parts to injected classes. In all, this facility supports the transformation challenges outlined in the previous section nicely.

5.2 JetBrains MPS

Syntax Since no grammar is used, the extending language can simply create new language constructs, some of them literally extending concepts from the base language. In cases where ambiguity would arise in a grammar-based system, MPS requires the language user to decide which of the alternatives should be used when the program is input. As a consequence, the tree, when it is created, is never ambiguous. A language in MPS can extend any number of languages. In particular, a program written with MPS can be expressed in any number of languages without first defining a composed language explicitly. This means that, for example, a set of extensions qto C can be developed independently, and users can then decide ad hoc which extensions they want to use in their program, as long as the extensions are semantically compatible. We discuss this problem in the transformation section. From a syntactic perspective, all modularization and composition approaches are supported by MPS without any limits.

Constraint/Type Systems 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 in language extensions, MPS supports so-called overloaded operation containers. This mechanism supports plugging in additional typing rules simply by adding them to the language definition. For example, binary operators in the C implementation are implemented with this mechanism, defining typing rules for the primitive types available in C. The language extension that adds types with physical units simply adds additional typing rules for the case when the left and/or right argument of a binary operator in a type with a physical unit.

Transformation 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, XML or C). All other "generations" happens by transformation rules that work on the projected tree. So for example, the concepts in a state machine extension to C are transformed back to the concepts in the C language it extends. So assimilation for language extension is supported naturally. Overwriting existing generators is also possible by modularizing generator is a suitable way and then using generator priorities to make sure the overriding generator runs before the overridden one. It is also possible to define hooks in generators, even if the approach to doing to is not really sat-

isfying. A generator can generate program elements whose only purpose is to be transformed further in subsequent generator stages. 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 it, 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 when the languages are used together in a fragment, not only when the transformation fails.

5.3 SDF/Spoofax

TODO►EV◄

TODO►Explain why we don't use interpretation◄ TODO►Table that compares the four approaches reg crit above◄ TODO►Check illustrations: do I explain the notation?◄

6. Related Work

7. Conclusion