Language Modularity, Reuse and Composition

Markus Voelter
independent/itemis
voelter@acm.org

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

This paper presents a classification of approaches for language modularization, reuse and composition. We identify 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. Decent support for language modularization and composition allows for programming environments where languages addressing different concerns of a system can be flexible combined into a very effective integrated development environment for a specific class of software systems.

1. Introduction and Motivation

Traditionally, programmers use general purpose languages (GPLs) for developing systems. "general-purpose" refers to the fact that they can be used for any programming task. They are Turing complete, and provide means to build custom abstractions using classes, higher-order functions, or logic predicates, depending on the particular language. Traditionally, a complete software system is implemented using a single GPL, plus a number of configuration files. However, more recently this has started to change; systems are built using a multitude of languages.

One reason is the rising level of sophistication and complexity of execution infrastructures. For example, web applications consist of business logic on the server, a database backend, business logic on the client as well as presentation code on the client, most of these coming with their own set of languages. A particular language stack could use Java, SQL, JavaScript and HTML. A web application is implemented by using these languages together. The second reason driving polyglot programming is increasing popularity of domain-specific languages (DSLs), specialized, often small languages that are optimized for expressing programs in a particular application domain. Such an application domain may be a technical domain, e.g. database querying with SQL, or an actual business domain, such as insurance contracts or 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 encountered in those domains. Using custom linguistic abstractions makes the code more concise, more accessible to formal analysis, verification, transformation and optimization, and possibly usable by non-programmer domain experts.

The use of polyglot programming raises the question how the syntax, semantics, and IDE support of the various languages is integrated. Especially syntactic integration has traditionally been very hard [15] and hence is often not supported for a particular combination of languages. Program parts expressed in different languages reside in different files. References among "common" things in these different program parts are implemented by using agreed-upon identifiers that must be used consistently. For some combinations of languages, the IDE may be aware of the "integration by name" and check the consistency. In some rare cases, syntactic integration between specific pairs of languages has been built, for example, embedded SQL in Java [2].

However, building specialized integrations between two languages is very expensive, especially if IDE support like code completion, syntax coloring and static error checking or refactoring is to be provided as well. So this is done only for combinations of very widely used languages, if at all. Building such an integration between Java and a company-specific DSL for financial calculations is infeasible. A more systematic approach for language modularization, reuse and composition (LMR&C) is required. Once available and supported by tools, this will lead to a new perspective on programming, taking advantage of the hitherto untapped potential for reuse. Languages can be extended as necessary, DSLs can be embedded in general purpose languages, and the distinction between programming and modeling, which creates

significant issues for model-driven development today, may go away completely [25].

The contribution of this paper is a systematic approach for characterizing modularized languages and their IDEs. We identify four different composition approaches which we introduce in Section 3: referencing, extension, reuse and embedding. For each of these approaches we provide a concise description, a formal definition, usage scenarios and an example. Section 4 describes challenges that need to be addressed when implementing the particular composition approach, and Section 5 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 as examples. To provide foundation for our discussion, the next section (Section 2) introduces terminology and concepts.

Our classification is useful because it provides a clearly defined vocabulary for LMR&C. Tools can be characterized regarding their support for any of the modularization approaches. DSL designers can use the vocabulary to discuss tradeoffs between the various approaches as they build DSLs. Researchers can use the vocabulary as a means of describing how novel approaches address the challenges outlined in this paper.

1.1 Challenges in language modularization

For each of the LMR&C approaches we discuss implementation challenges in three areas. Many language *syntax description* formalisms, such as LL(k), are not closed under composition. Combining two valid grammars may result in an invalid, ambiguous grammar. To support LMR&C languages have to be described with formalisms that support combinations of independently developed grammars. The languages used for describing *type systems* and constraints have to be modular as well, supporting enhancement or combination with other type system definitions. Finally, reusable languages may already come with *transformations*. These may have to be reused, combined or extended as well.

1.2 Example Tools

In this paper we use three different tools to illustrate support for LMR&C. We have chosen Xtext, Spoofax and MPS because of their different approaches to language implementation and their varying support for LMR&C. The three example tools are language workbenches [11], which derive many of the ingredients necessary to edit and process programs from language definitions. Specifically, they all derive language-specific IDEs. We consider IDE support important for efficiently working with languages, which is why we do not consider pure grammar definition or parser generation tools such as LEX/YACC, ANTLR or SDF in this paper. This requirement also rules out internal DSLs (we will come back to this in Related Work). Xtext, MPS and Spoofax all provide IDE support for those LMR&C 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 and static error checking for programs written in the combined languages.

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

Xtext As part of the Eclipse Modeling framework, Xtext[1] 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 LMR&C is limited, partly because of its reliance on the antlr parser generator, which is ll(k). The main reason for including it in this discussion is that it is widely used — it can be considered the industry standard tool for the definition of textual DSLs and IDEs — and that it generates very complete IDE support.

Spoofax Spoofax[2] is an Eclipse-based language workbench built on the SDF syntax definition formalism and Stratego transformation language [**?**]. While it is not yet widely used, several non-trivial languages and IDEs have been built with it [20, 12]. We include it in the discussion because of its extensive support for LMR&C.

MPS MPS[3] is developed by JetBrains. It is fundamentally different from the other two tools because it uses projectional editing. No parser is involved, changes to a program's textual representation directly change the underlying program tree, so many of the challenges of grammar composition do not exist. Despite the fact that MPS is not used widely yet, we have included in this discussion because of its very good support for LMR&C, and its use of projectional editing. An extensive, detailed description of how MPS implements the four composition approaches discussed in this paper is available in [24].

2. Concepts and Terminology

2.1 Programs, Languages, Domains

In this paper we will be talking about *programs*, *languages*, and *domains*, so we explain these terms here. We start by considering 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

[1] http://eclipse.org/Xtext

[2] http://spoofax.org

[3] http://jetbrains.com/mps

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. 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 refigerator cooling 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_{vcool} is the set of cooling programs for the particular refrigerator 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. We will come back to this approch in the section on language extension (Section 3.2).

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 (e.g. regarding program size or complexity) 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 in doing so. This is where DSLs have an advantage.

The fact that DSLs are 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, support for LMR&C is required.

2.2 Programs as Trees of Elements

Programs are represented in two ways: concrete syntax and abstract syntax. 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. 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.

While concrete syntax modularization and composition can be a challenge (as discussed in Section 4), 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* that usually reference each other. 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 relation-

ships 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. 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 not depend on other languages. An independent language l can be defined as a language for which the following hold:

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

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

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

An *independent fragment* is one where all references stay within the fragment (4).

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

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

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

$$\forall c \in Cdn_f \mid lo(c.parent) = lo(c.child) = l \quad (6)$$

3. Classification

We now have all the ingredients to discuss the various approaches to LMR&C. 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. Fig. 2 shows the relationships between fragments and languages in these cases. We consider these two criteria to be essential for the following reasons.

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

3.1 Language Referencing

Language referencing enables *homogeneous* fragments with cross-references among them, using *dependent* languages (Fig. 3).

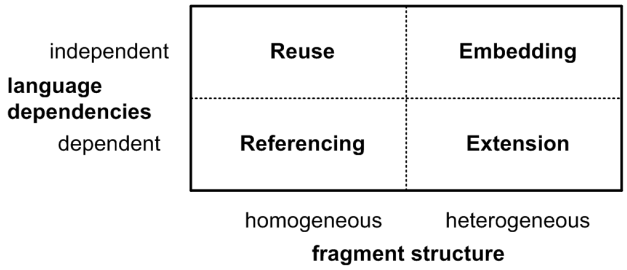

Figure 1. We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies. The dependencies dimension captures whether the languages have to be designed specifically for a specific composition partner or not. Fragment structure captures whether the composition approach supports mixing of the concrete syntax of the composed languages or not.

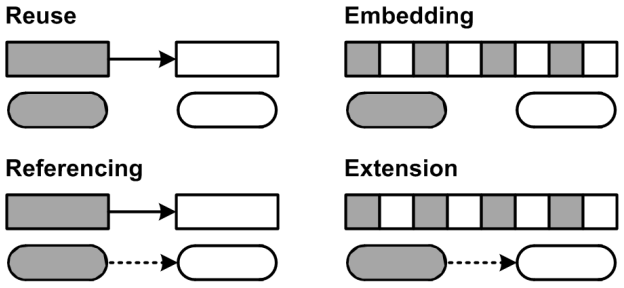

Figure 2. The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two composed languages.

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

$$\forall r \in Refs_{l_2} \mid lo(r.from) = l_2 \ \wedge \ lo(r.to) = (l_1 \vee l_2) \quad (7)$$

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

3.1.1 Use Cases and Examples

A domain D can be composed from different concerns. To describe a complete program $p \in P_D$, the program needs to address all these 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 supports addressing all concerns of D in one 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. The latter alternative has the advantage that different stakeholders can modify "their" concern independently from 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.

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 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 is implemented as a separate language with references between them. H and A are separated because H is defined by product management, whereas A is defined by thermodynamics engineers. 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.

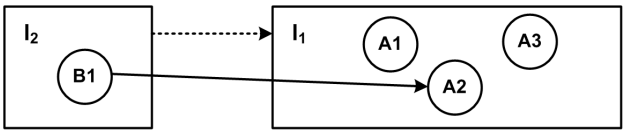

Figure 3. Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

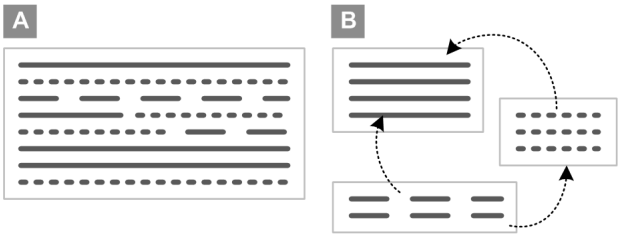

Figure 4. Part A shows an integrated DSL where the various concerns, represented by different line styles in the boxes, are covered by a single integrated language and reside in one fragment. Part B shows several viewpoint languages and program fragments, each covering a single concern. Arrows in Part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

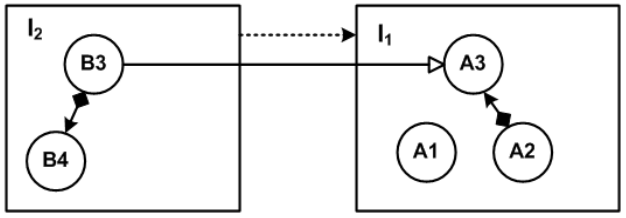

Figure 5. Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

3.2 Language Extension

Language extension enables *heterogeneous* fragments with *dependent* languages (Fig. 5). 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 some of the concepts in l_2 will inherit from concepts in l_1:

$$\exists i \in Inh(l_2) \mid i.sub = l_2 \ \wedge \ i.super = l_1 \tag{8}$$

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

$$\forall e \in E_f \mid lo(e) = (l_1 \vee l_2) \tag{9}$$

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

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = (l_1 \vee l_2) \wedge lo(co(c.child)) = (l_1 \vee l_2) \tag{10}$$

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 supports calling one language from another, like calling C from Perl or Java, is not language extension; rather it is a form of language referencing. The fragments remain homogeneous.

3.2.1 Use Cases and Examples

Hierarchical Domains Domains are naturally organized in hierarchies Fig. 6). At the bottom we find the most general domain D_0. It is the domain of all possible programs

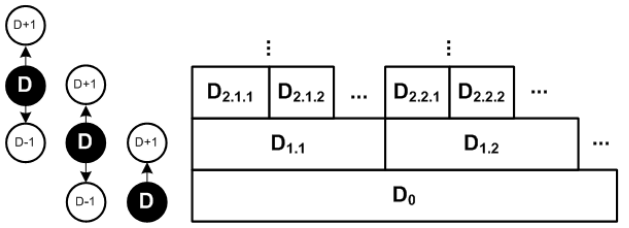

Figure 6. Domain hierarchy. Domains with higher index are called subdomains of domains with a lower index (D_1 is a subdomain of D_0). We use just D to refer to the current domain, and D_{+1} and D_{-1} to refer to the relatively more specific and more general ones.

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 (obviously not useful). Languages designed 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 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 *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. As an example consider embedded programming. The C programming language is typically used as the GPL for D_0 in this case. Extensions for embedded programming include state machines, tasks or data types with physical units. Language extensions for the subdomain of real-time systems may include ways of specifying deterministic scheduling and worst-case execution time. For the avionics subdomain support for remote communication using some of the bus systems used in avionics could be added. For examples of extensions of C for embedded programming see [22] and [6].

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).

3.3 Language Reuse

Language reuse (Fig. 7) 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

$$\exists r \in \mathit{Refs}_{f_2} \mid \mathit{fo}(r.\mathit{from}) = f_2 \wedge \mathit{fo}(r.\mathit{to}) = (f_1 \vee f_2) \tag{11}$$

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

One way of realizing dependent fragments while retaining independent languages is using an adapter language (cf. the Adapter pattern) l_A where l_A *extends* l_2 and

$$\exists r \in \mathit{Refs}_{l_A} \mid \mathit{lo}(r.\mathit{from}) = l_A \ \wedge \ \mathit{lo}(r.\mathit{to}) = l_1 \tag{12}$$

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.

3.3.1 Use Cases and Examples

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 the dependency direction to

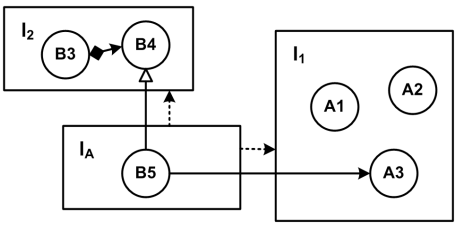

Figure 7. Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

$data \rightarrow ui$ doesn't solve the problem either, because this would go against the model-view-controller pattern, which states that the UI may have dependencies to the data, but not vice versa.

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.

3.4 Language Embedding

Language embedding enables *heterogeneous* fragments with *independent* languages (Fig. 7, but with a containment link between $B5$ and $A3$). It is similar to reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now embed instances of concepts from l_2 in a fragment f expressed with l_1, so

$$\forall c \in Cdn_f \mid lo(co(c.parent)) = l_1 \wedge lo(co(c.child)) = (l_1 \vee l_2) \quad (13)$$

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

$$\exists c \in Cdn_{l_A} \mid lo(c.parent) = l_A \wedge lo(c.child) = l_1 \quad (14)$$

3.4.1 Use Cases and Examples

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

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

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

4. Implementation Challenges and Solutions

The previous section has introduced four LMR&C approaches. In this section we investigate the consequences of these approaches for the composition of concrete syntax, type system and transformation definitions.

4.1 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.

4.2 Type Systems

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. For example, a compressor has a property `active` that controls if it is 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 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.).

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.

4.3 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.

Referencing Three cases have to be considered. The first one (Fig. 8 A) 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.

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.

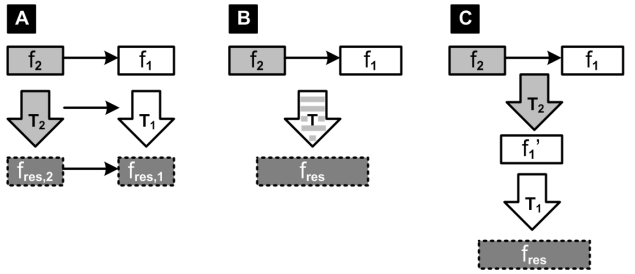

Figure 8. Three alternatives of handling the transformations for language referencing: (A) two separate, dependent, single-source transformations, (B) a single multi-sourced transformation and (C) a preprocessing transformation that changes the referenced fragment in a way specified by the referencing fragment, then reusing existing transformations for the referenced fragment (Notation for this and the following illustrations: boxes are fragments, fat arrows are transformations, thin arrows are dependencies and shading represents languages.)

The second case (Fig. 8 B) 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. 8 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.

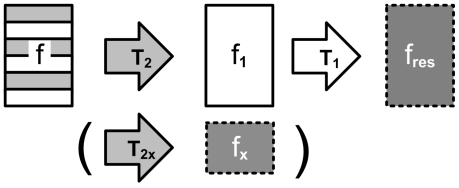

Figure 9. Transformation for language extensions usually happens by assimiliation, i.e. generating code in the host language from code expressed in the extension language. Optionally, additional files are generated, often some configuration files.

Extension 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. 9). 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.

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.

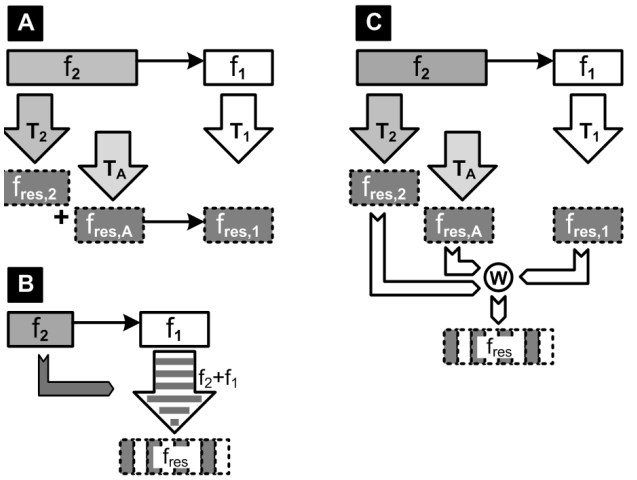

Figure 10. Transformation in the case of reuse comes in three flavors. reuse of existing transformations for both fragments plus generation of adapter code (A), composing transformations (B) or generating separate artifacts plus a weaving specification (C).

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.

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

There are three cases to consider. The first one, illustrated in Fig. 10 A, 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. 10 B, 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. The third case, Fig. 10 C, leaves composition to the target languages. We generate three different independent, homogeneous fragments, and 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.

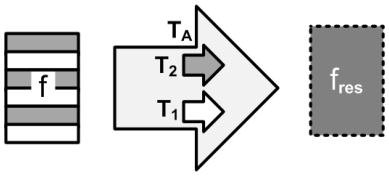

Figure 11. In transforming embedded languages, a new transformation has to be written if the embedded language does not come with a transformation for the target language of the host language transformation. Otherwise the adapter language can coordinate the transformations for the host and for the emebedded languages.

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. 11 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

In this section we look discuss to what extent our the example tools introduced in Section 1.2 support the LMR&C approaches and the implementation challenges.

5.1 Eclipse Xtext

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/Type Systems 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[4]. 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.

Transformation 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.

5.2 JetBrains MPS

Syntax 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 introduction on the language level. Attributes are typically used for adding ”meta data” such as documentation, presence conditions in product line engineering [23] or requirements traces.

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, 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.

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 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.

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

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.

5.3 Spoofax

Syntax Syntax definition in Spoofax is based on the SDF2 Syntax Definition Formalism [19]. Since SDF uses the Scannerless Generalized-LR parsing algorithm, the full class of context-free grammars is supported, which is the only grammar class closed under composition, unlike the restricted grammar classes of other parser generators. Furthermore, the parser does not use a separate lexical analysis phase, but rather considers individual tokens as characters, which entails that it supports combination of languages with a different lexical syntax, as demonstrated, for example, in the syntax definition for AspectJ [5]. Thus, SDF supports both extension and embedding of languages.

Analysis and Transformation Semantic analysis, transformation, and code generation in Spoofax are defined with the Stratego transformation language [21, 3], which is based on the paradigm of term rewriting with programmable strategies. Basic transformations are defined with rewrite rules and can be combined into composite transformations using strategies. Thus, basic rules defining type analysis, static semantic constraints, desugarings, and other transformations for separate language components can be composed into transformations for language compositions. Stratego provides aspects for overriding and adapting transformation strategy definitions [**?**].

6. Related Work

The contribution of this paper is the systematic definition and classification of language modularization approaches, as well as the challenges and solution involved regarding syntax definition, type systems, transformations and IDE support. The idea of language modularization and composition itself is not new, however.

In their paper "When and How to design DSLs" [16], Mernik et. al. discuss various aspects of DSL design. Among other things, they also describe a number of modularization approaches, among them extension and restriction. In our paper, we provide a clearer definition and discuss the challenges and solution approaches. Mernik et al. also propose Piggybacking and Pipelining as ways to reuse existing generators or interpreters. While these approaches are certainly useful, we don't include them in our discussion here because they don't *compose* languages — they just chain their translation.

Incremental Extension of Languages was first popularized in the context of Lisp, where definition of language extensions to solve problems in a given domain is a well-known approach. Guy Steele's Growing a Language keynote explains the idea well [14]. The paper on Xoc [6] describes the idea of extending C incrementally, albeit without extending a corresponding IDE. Sergey Dmitriev discusses the idea of language and IDE extension in his article on Language Oriented Programming [7], which uses MPS as the tool to achieve the goal.

Macro Systems support the definition of additional syntax for existing languages. Macro expansion maps the new syntax to valid code in the extended language, and this mapping is expressed with host language code instead of a separate transformation language.They differ with regard to degree of freedom they provide for the extension syntax, and whether they support extensions of type systems and IDEs. The most primitive macro system is the C preprocessor which performs pure text replacement during macro expansion. The Lisp macro system is more powerful because it is aware of the syntactic structure of Lisp code. An example of a macro system with limited syntactic freedom is the The Java Syntactic Extender [1] where all macros have to begin with names, and a limited set of syntactic shapes is supported. In OpenJava [18], the locations where macros can be added is limited. More fine-grained extensions, such as adding a new operator, are not possible.

Language Cascading refers to a form of language combination where a program expressed in languge l_1 is translated into a program expressed in language l_2. Essentially this is what every code generator or compiler does; the languages themselves are not related in any way except through the transformation engine, which is why we don't consider this as an example of language modularization and composition. An example of this approach is KHEPERA [10]

Full Language Extension and Composition refers to the case where arbitrary syntax can be added to a host language, and type systems and IDEs are aware of the extension. This paper classifies four approaches for doing this. Existing implementations exist. For example, Bravenboer and Visser describe how SQL can be embedded into Java to prevent SQL injection attacks [2]. The same authors discuss library-based language extension and embedding in [4]. A more recent publication [9] also based on SDF introduces SugarJ, which supports library based languages extension. [8] adds IDE support.

Internal DSLs are languages embedded in general purpose host languages. Suitable host languages are those that provide a flexible syntax, as well as meta programming facilities to support the definition of new abstractions with a custom concrete syntax. For example [13] describes embedding DSLs in Scala. In this paper we don't address internal DSLs, because IDE support for the embedded languages is

not available in these cases, and we consider IDE support for the composed languages essential.

7. Discussion and Future Work

The modularization approaches discussed in this paper are supported to various degrees by the example tools. In this section we compare those approaches to established module and component systems as a way of identifying what is still missing. According to Szyperski's book Component Software [17], components can essentially be characterized as follows: (1) A component defines interfaces that provide access to its behavior. The actual implementation is hidden. Interfaces also specify that behavior to some extent, for example using pre- and post-conditions or protocol state machines. (2) A component makes all its context dependencies explicit, for example, by referring to the interfaces it makes use of. (3) Components also often work on the binary level, where the source code is not necessary to use or extend a component.

None of the tools support language interfaces. It is not even clear what a language interface would be, although it is clear that it would have to export not just the syntax, but also some of the type system rules and transformations. There is no hiding of internal structure. All tools support declaration of context dependencies in the form of references to extended or used languages. However, because of the lack of interfaces, it is not possible to exchange a referenced language for another one that provides the same interfaces. Support for binary reuse exists to some extent. For example, in MPS all the LMR&C approaches work with the referenced languages packaged in JAR files.

Essentially, the implementations of the LMR&C approaches is comparable to white box object-oriented frameworks where all classes are available for use and extension and we cannot even have the equivalent of Java's `final` which prevents the overriding of some methods or the extension of classes.

References

[1] J. Bachrach and K. Playford. The java syntactic extender (jse). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2001.

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

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCP*, 72(1-2):52–70, 2008.

[4] M. Bravenboer and E. Visser. Designing syntax embeddings and assimilations for language libraries. In *MoDELS*, pages 34–46, 2007.

[5] M. Bravenboer, Éric Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *OOPSLA*, pages 209–228, 2006.

[6] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *asplos*, pages 244–254, 2008.

[7] S. Dmitriev. Language oriented programming: The next programming paradigm, 2004.

[8] S. Erdweg, L. C. L. Kats, Rendel, C. Kastner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *GPCE*, 2011.

[9] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. In *OOPSLA*, 2011.

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

[11] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[12] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *OOPSLA*, 2011.

[13] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE*, pages 137–148, 2008.

[14] G. L. S. Jr. Growing a language. *lisp*, 12(3):221–236, 1999.

[15] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *OOPSLA*, pages 918–932, 2010.

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

[17] Szyperski and C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Number ISBN 0-201-17888-5. Addison-Wesley, 1998.

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

[19] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

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

[21] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, pages 13–26, 1998.

[22] M. Voelter. Embedded software development with projectional language workbenches. In *MoDELS*, pages 32–46, 2010.

[23] M. Voelter. Implementing feature variability for models and code with projectional language workbenches. 2010.

[24] M. Voelter. Language and ide development, modularization and composition with mps. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2011*, LNCS. Springer, 2011.

[25] M. Völter. From programming to modeling - and back again. *IEEE Software*, 2011.