Noname manuscript No.
(will be inserted by the editor)

Language and IDE Modularization and Composition with MPS

Markus Voelter

Received: date / Accepted: date

Abstract Language and IDE modularization and composition is an important building block for working efficiently with DSLs. Historically, this has been a challenge because many grammar formalisms are not closed under composition, hence syntactic composition of languages is challenging. Composing static and dynamic semantics can also be hard, at least in the general case. Finally, a lot of existing work does not consider IDEs for the composed and extended languages. In this paper, I will show how the projectional language workbench JetBrains MPS solves most of these issues. The main part of the paper is a set of extensive examples that show the various kinds of extension and modularization. The last section contains an evaluation that identifies the strong and weak aspects of modularization, composition and extension in MPS, and suggests a couple of improvements.

Keywords DSLs · language composition · language extension · JetBrains MPS · Language Workbench

1 Introduction

Traditionally, programmers use general purpose languages (GPLs) for developing software 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 has been 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.

Markus Voelter
Oetztaler Strasse 38, Stuttgart, Germany
E-mail: voelter@acm.org

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 implemented with their own set of languages. A particular language stack could use Java, SQL, JavaScript and HTML. The second reason driving polyglot programming is increasing popularity of domain-specific languages (DSLs). These are 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 a 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 (integrated development environment) support of the various languages can be integrated. Especially syntactic integration has traditionally been very hard [24] 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 [4].

However, building specialized integrations between two languages is very expensive, especially if IDE support like code completion, syntax coloring, static error checking, refactoring or debugging 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 and IDE modularization and compositionis required. Such an approach has to address the following concerns:

- The concrete and the abstract syntax of the two languages have to be composed. This may require the embedding of one syntax into another one. This, in turn, requires modular syntax definitions.
- The static semantics (i.e., the constraints and the type system) have to be integrated. For example, in the case of language extension, new types have to be "made valid" for existing operators.
- The execution semantics have to be combined as well. In practice, this may mean mixing the code generated from the composed languages, or composing the generators.

– Finally, the IDE that provides code completion, syntax coloring, static checks and other relevant services has to be extended and composed.

In this paper we focus on JetBrains MPS [1] as a means of demonstrating language composition approaches. MPS is a projectional editor, so no grammars or parsers are used. Instead, editing gestures *directly* modify the abstract syntax tree, and the representation on the screen is rendered, or projected, from the changing tree. As we show in this paper, this simplifies the syntactic aspect of language composition. Also, MPS been designed to be used for developing sets of integrated languages, and not just one or more standalone languages. This is exemplified by its extensible transformation and type checking frameworks.

1.1 Contribution and Structure of the paper

Language composition is the integration of language modules regarding syntax, static semantics, execution semantics and the IDE.

In this paper we make the following contributions. First, we identify four different composition approaches (referencing, extension, reuse and embedding) and classify them regarding dependencies and syntactic mixing. Second, we demonstrate how to implement these four approaches with JetBrains MPS. While other, parser-based approaches can do language composition to some extent as well, it is especially simple to do with projectional editors. So our third contrinbution is an illustration of the benefits of using projectional editors in the context of language composition, since MPS is an example projectional editor.

The paper is structured as follows. In Section 1.3 we define a set of terms and concepts used in this paper. Section 1.4 outlines the various kinds of language and IDE modularization and compositiondiscussed in this paper, and provides rationale why we discuss those kinds, and not others. Then we describe how projectional editors work in general, and how MPS works specifically (Section 2) and develop the core language which acts as the basis for the extension and composition examples (Section 3). This section also serves as a very brief tutorial on language definition in MPS. The main part of the paper, the implementation of the various extension and composition approaches, is discussed in Section 4. We look at other contemporary language workbenches as well as general related work in Section **??**. Finally, Section 5 discusses what works well and at what could be improved in MPS with regards to extension and composition.

1.2 Additional Resources

The example code developed for this tutorial can be found at github.com and works with MPS 2.0:

[1] http://www.jetbrains.com/mps/

`https://github.com/markusvoelter/MPSLangComp-MPS2.0`

A set of recorded demos (90 minutes in total) that walk through all the example code is available on Youtube. The initial video is here:

`http://www.youtube.com/watch?v=lNMRMZk8KBE`.

The others are either suggested by Youtube, or you can find them by searching for *Language Modularization and Composition with MPS (Part X)*, where X is between 1 and 8.

Note that this paper is not a complete MPS tutorial. MPS is very deep and powerful, so we have to focus on those aspects that are essential for language and IDE modularization and composition. We refer to the Language Workbench Competition (LWC 11) MPS tutorial for details:

`http://code.google.com/p/mps-lwc11/wiki/GettingStarted`

1.3 Terminology

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. MPS is a projectional editor.

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 may 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: $co(element) \Rightarrow concept$. Similarly we define the *language-of* function lo to return the language in which a given concept is defined: $lo(conept) \Rightarrow language$. Finally, we define a *fragment-of* function fo that returns the fragment that contains a given program element: $fo(element) \Rightarrow fragment$.

We also define the following sets of relations between program elements. Cdn_f is the set of parent-child relationships in a fragment f. Each $c \in C$ has the properties *parent* and *child*. $Refs_f$ 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 [29] 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 concern in language and IDE modularization and composition 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:

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

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

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

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

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

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

$$\forall e \in E_f \mid lo(e) = l \tag{5}$$

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

In this paper we consider the semantics of a language l_1 to be defined via a *transformation* that maps a program expressed in l_1 to a program in another language l_2 that has the same *observable behavior*. The observable behavior can be determined in various ways, for example using a sufficiently large set of test cases. A discussion of alternative ways to define language semantics is beyond the scope of this paper. In particular, we also do not discuss interpreters as an alternative to transformations. However, in our experience, transformations are by far the most used approach for defining semantics, so the focus on transformations is not a significant limitation in practice.

The paper emphasizes *IDE* modularization and composition in addition to *language* modularization and composition. In the context of this paper, when referring to IDE services, we mean syntax highlighting, code completion and static error checking. Other concerns are relevant in IDEs, including refactoring, quick fixes, support for testing, debugging and version control integration. While all of these are supported by MPS in a modular and composable way, we do not discuss those aspects in this paper to keep the paper reasonable is length.

1.4 Classification of Composition Approaches

In this paper we have identify the following four modularization and composition approaches: referencing, extension, reuse and embedding. Below is an intuitive description of each approach; stricter definitions follow in the rest of the paper.

Referencing refers to the case where two languages are composed by a language A referencing concepts defined in language B. The programs expressed in A and B are kept in separate homogeneous fragments (files), and only name-based references connect the programs. The referencing language has a direct dependency on the referenced language. An example for this case is a language that defines user interface forms for data structures defined by another language. The user interface language references the data structures defined in a separate program.

Extension also allows a dependency of the extending language to the extended languages (also called base language). However, in this case the code written in the two languages resides in a single, heterogeneous fragments, i.e. syntactic composition is required. An example would be the extension of Java or C with new types, operators or literals.

Reuse is similar to referencing in that the respective programs reside in separate fragments and only name-based references connect the programs. However, in contrast to referencing, no direct dependencies between the languages are allowed. An example would be a persistence mapping language that can be used together with *different* data structure definition languages. To make this possible, it cannot depend on any particular data definition language.

Embedding combines the syntactic integration introduced by extension with not having dependencies introduced by reuse. So independent languages can still be used in the same heterogeneour fragment. An examples includes the embedding of a reusable expression language into a DSL. Since neither of the two composed languages have direct dependencies, the same expression language can be embedded into *different* DSLs, and a specific DSL could integrate *different* expression languages.

As can be seen from the above descriptions, 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. We used these two criteria as the basis for this paper because we consider these two criteria to be essential for the following reasons.

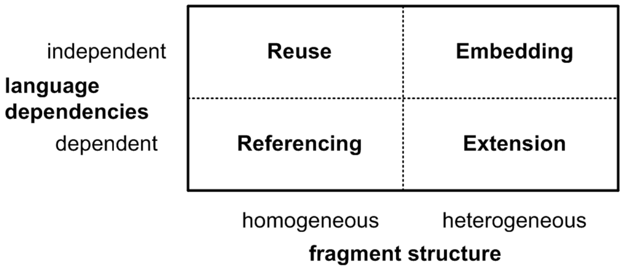

Fig. 1 We distinguish the four modularization and composition approaches regarding their consequences for fragment structure and language dependencies. The dependencies dimension captures whether the languages have to be designed specifically for a specific composition partner or not. Fragment structure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

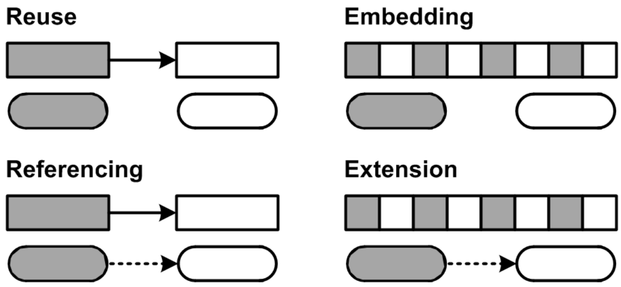

Fig. 2 The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

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 easily possible, though often desirable.

Other classification approaches have been proposed. In particular, in their paper "When and How to design DSLs" [33], Mernik et. al. propose a number of modularization approaches, among them extension and restriction. In the context of the classification proposed in our paper, restriction is a form of extension in the following sense: to restrict a language, we develop an extension that *prohibits the use of some language concepts in certain contexts*. We discuss this at the end of Section 4.2. Mernik et al. also propose Piggybacking and Pipelining as ways to reuse existing generators or interpreters. We don't include these approaches in our discussion here because they don't *compose* languages — they just chain their translation.

1.5 Case Study

In this paper we illustrate the language and IDE modularization and compositionapproaches with MPS. At the center is a simple `entities` language. We then build an additional language to illustrate language and IDE modularization and composition. Fig. 3 illustrates these additional languages. The `uispec` language illustrates *referencing* with `entities`. `relmapping` is an example of *reuse* with separated generated code. `rbac` illustrates reuse with intermixed generated code. `uispec_validation` demonstrates *extension* (of the `uispec` language) and *embedding* with regards to the expressions language.

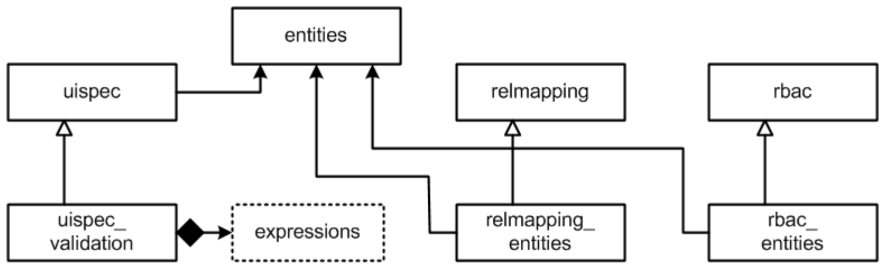

Fig. 3 `entities` is the central language. `uispec` defines user interface (UI) forms for the `entities`. `uispec_validation` adds validation rules, and composes a reusable expressions language. relmapping provides a reusable database mapping language, `relmapping_entities` adapts it to the `entities` language. rbac is a reusable language for specifying permissions; `rbac_entities` adapts this language to the `entities` language.

2 How MPS works

The JetBrains Meta Programming System [2] is a projectional language workbench available as Open Source software under the Apache 2.0 license. The term Language Workbench was coined by Martin Fowler in 2004 [15]. In his article he defines a language workbench as a tool with the following characteristics:

1. Users can freely define languages which are fully integrated with each other.
2. The primary source of information is a persistent abstract representation.
3. A DSL is defined in three main parts: schema, editor(s), and generator(s).
4. Language users manipulate a DSL through a projectional editor.
5. A language workbench can persist incomplete or contradictory information.

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

MPS exhibits all of these characteristics. MPS' most distinguishing feature is its projectional editor. This means that all text, symbols, and graphics are projected, and not parsed. Projectional editing is well-known from graphical modeling tools (UML, Entity-Relationship, State Charts). The model is stored independent of its concrete syntax, only the model structure is persisted, often using XML or a database. For editing purposes, graphical editors project the abstract syntax using graphical shapes. Users use mouse gestures and keyboard actions tailored to graphical editing to modify the abstract model structure directly. While the concrete syntax of the model does not have to be stored because it is specified as part of the language definition and hence known by the projection engine, graphical modeling tools usually also store information about the visual layout.

Projectional editing can also be used for textual syntax. However, since the projection looks like text, users expect editing gestures known from "real text" to work. For a projectional editor to be useful, it has to "simulate" interaction patterns known from real text. MPS achieves this quite well. How it does that is beyond the scope of this paper. The following is a list of benefits of the projectional editing approach:

- No grammar or parser is required. Editing directly changes the underlying structure. Projectional editors can handle unparseable code. Language composition is made easy, because it cannot result in ambiguous grammars [1].
- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions[3].
- Because projectional languages by definition need an IDE for editing (it has to do the projection!), language definition and extension always implies IDE definition and extension. The IDE will provide code completion, error checking and syntax highlighting for all languages, even when they are composed.
- Because the model is stored independent of its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints of the overall program can be stored in one model, but editing can still be viewpoint-specific. It is also possible to store out-of-band data (i.e. annotations on the core model/program. Examples of this include documentation, pointers to requirements (traceability) or feature dependencies in the context of product lines.

Projectional editors also have drawbacks. The first one is that editing the projected representation as opposed to "real text" needs some time to get used to. Without specific customization, every program element has to be selected from a drop-down list to be "instantiated". However, MPS provides editor

[3] Intentional's Domain Workbench has demonstrated this repeatedly, for example in [44]. As of 2012, MPS can do text, symbols (such as big sum signs or fraction bars) and tables. Graphics will be supported in 2013.

customizations to enable an editing experience that resembles modern IDEs that use automatically expanding code templates. This makes editing quite convenient and productive in all but the most exceptional cases. The second drawback is that models are not stored as readable text, but rather as an XML-serialized abstract syntax tree. Integrating XML files with an otherwise ASCII-based development infrastructure can be a challenge. MPS addresses the most critical aspect of this drawback by supporting diff and merge on the level of the projected concrete syntax. Finally, MPS is proprietary in the sense that it is not based on any industry standards. For example, it does not rely on EMF or another widely used modeling formalism. However, since MPS' meta meta model is extremely close to EMF Ecore, it is trivial to build an exporter that exports ASTs as an EMF model. Also, all other language workbenches also do not support portability of the *language definition* beyond the abstract syntax — which is the simplest aspect in terms of implementation effort.

MPS has been designed from the start to work with sets of integrated languages. This makes MPS particularly well suited to demonstrate language and IDE modularization and composition techniques. In particular, the following three characteristics are important in this context:

Composable Syntax: Depending on the particular composition approach used, syntactic composition of the languages is required. In traditional, grammar-based systems, combining independently developed grammars can be a problem: many grammar classes are not closed under composition, and various invasive changes (such as left-factoring or redefinition of terminals or non-terminals), or unwanted syntactic escape symbols are required [24]. As we will see, this is not the case in MPS. Arbitrary languages can be combined syntactically.

Extensible Type Systems: All of the composition techniques require some degree of type system extension or integration. MPS' type system specification is based on decarative type system rules that are executed by a solver. This way, additional typing rules for additional language constructs can be defined without invasively changing the existing typing rules of the composed languages.

Modular Transformation Framework: Transformations can be defined separately for each language concept. If a new language concept is added via a composition technique, the transformation for this new concept is modular. If existing transformation should be overridden or a certain program structure must be treated specially, a separate transformation for these cases can be written, and, using generator priorities, it can be made sure that it runs *before* an existing transformation. This way, existing transformations can be enhanced or overridden in a modular way.

The MPS-based examples for the language composition techniques discussed in this paper will elaborate on these characteristics. This is why for each technique, we discuss structure and syntax, type system and transformation concerns.

3 Implementing a DSL with MPS

This section illustrates the definition of a language with JetBrains MPS. Like other language workbenches, MPS comes with a set of DSLs for language definition, a separate DSL for each language aspect. Language aspects include structure, editor, type systems, generators as well as things like quick fixes or refactorings. MPS is bootstrapped, so these DSLs are built with MPS itself.

At the center of the language extensions we will build later, we use a simple `entities` language (some example code is shown below). *Modules* are root nodes. They live as top-level elements in *models*. Referring back to the terminology introduced in the DSL design paper [**?**], root nodes (and their descendants) are considered *fragments*, while the models are partitions (actually, they are XML files).

```
module company
  entity Employee {
    id : int
    name : string
    role : string
    worksAt : Department
    freelancer : boolean
  }
  entity Department {
    id : int
    description : string
  }
```

■ *Structure and Syntax.* Language definition starts with the abstract syntax, called *concepts* in MPS. Fig. 4 shows a UML diagram of the structure of the `entities` language. Each box represents a language concept.

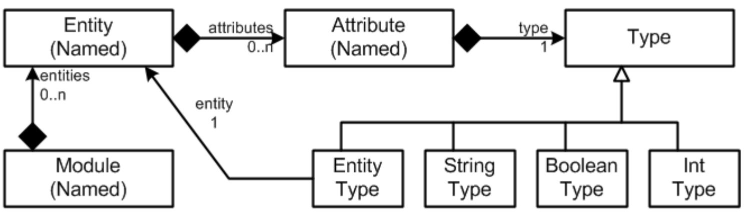

Fig. 4 The abstract syntax of the entities language. Entities have attributes which have types and names. `EntityType` extends `Type` and references `Entity`. This adapts entities to types (cf. the Adapter pattern [16]). Concepts like `EntityType` which have exactly one reference are called smart references and are treated specially by MPS: the code completion menu shows *the possible targets* of the reference directly, instead of first instantiating the reference concept and then selecting the target.

The following code shows the definition of the `Entity` concept[4]. `Entity` extends `BaseConcept`, the root concept, similar to `java.lang.Object` in Java. It implements the `INamedConcept` interface to inherit a `name` field. It declares a list of children of type `Attribute` in the `attributes` role. A concept may also have references to other concepts (as opposed to children).

```
concept Entity extends BaseConcept implements INamedConcept
  is root: true
  children:
    Attribute attributes 0..n
  references:
    << ... >>
```

Editors in MPS are based on cells. Cells are the smallest unit relevant for projection. Consequently, defining an editor consists of arranging cells and defining their content. Different cell types are available to compose editors. Fig. 5 explains the editor for `Entity`. The editors for the other concepts are defined similarly.

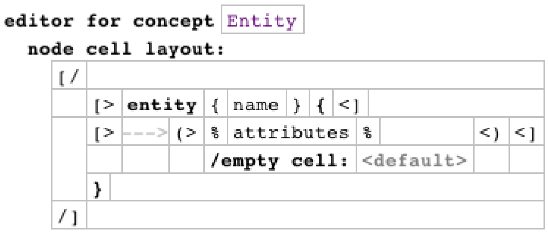

Fig. 5 The editor for `Entity`. The outermost cell is a vertical list. In the first line, we use a horizontal list that contains the "keyword" `entity`, the value of the `name` property and an opening curly brace. In the second line we use indentation and a vertical arrangements of the contents of the `attributes` collection. Finally, the third line contains the closing curly brace.

■ *Type System.* The MPS type system engine uses unification. Language developers specify type equations and the unification engine tries to assign values to the type variables so that all equations are satisfied. This is similar to solving a set of equations in mathematics. Consider the following three equations:

```
(1) 2 * x == 10          (2) x + x == 10          (3) x + y == 2 * x + 5
```

[4] In addition to `properties`, `children` and `references`, concept definition can have more characteristics such as `concept properties` or `concepts links`. However, these are not needed for this example, so we don't show them here. The code above shows all the characteristics used in this example

This set of equations can be solved by `x := 5, y := 10`. The MPS type system engine works the same way, but the domain is types instead of integers. Type equations also do not just contain equations (:==:), but also equations with subtyping and other relationships.

For the `entities` language, we specify two simple typing rules. The first one specifies that the type of a `Type` (such as `int` or `string`) is a clone of themselves.

```
rule typeof_Type for Type as t {
    typeof(t) :==: t.copy;
}
```

The only other typing rule is an equation that defines the type of the attribute as a whole to be the type of the attribute's `type` property, defined as `typeof(attribute) :==: typeof(attribute.type);`. No other typing rules apply in this simple entities language. As we have said in Section 2, language developers only have to specify the typing rules. MPS type system engine applies the rules to all applicable program elements, solves the resulting set of type equations, and if equations run into contradictions, this is annotated on the offending program element as a type error.

■ *Generator.* From entity models we generate Java Beans. Since Java is available in MPS as the BaseLanguage, the generation is actually a model-to-model transformation: from the `entities` model we generate a Java model. MPS supports several kinds of transformations. The default case is the template-based transformation which uses the concrete syntax of the target language to specify model-to-model transformations. Alternatively, one can use the node API to manually construct the target tree. Finally, the `textgen` DSL is available to generate ASCII text (at the end of the transformation chain). Throughout this paper we use the template-based approach.

Template-based generators in MPS consist of two main building blocks: mapping configurations and templates. Mapping configurations define which elements are processed with which templates. For the `entities` language, we need a *root mapping rule* and *reduction rules*. Root mapping rules can be used to create new top-level artifacts from existing top-level artifacts (they map fragments to other fragments). In our case we generate a Java class from an entity. Reduction rules are in-place transformations. Whenever the engine encounters an instance of the specified source concept somewhere in a model tree, it removes that source node and replaces it with the result of the associated template. In our case we have to reduce the various types (`int`, `string`, etc.) to their Java counterparts. Fig. 6 shows a part of the mapping configuration for the `entities` language.

```
root mapping rules:
  [concept          Entity   ] --> map_Entity
  [inheritors       false    ]
  [condition        <always> ]
  [keep input root  true     ]

reduction rules:
  [concept     IntType  ] --> <T int T>
  [inheritors  false    ]
  [condition   <always> ]

  [concept     EntityType] --> <T ->$[Double] T>
  [inheritors  false     ]
  [condition   <always>  ]
```

Fig. 6 The mapping configuration for the `entities` language. The root mapping rule for `Entity` specifies that instances of `Entity` should be transformed with the `map_Entity` template (which produces a Java class). The reduction rules use inline templates, i.e. the template is embedded in the mapping configuration. For example, the `IntType` is replaced with the Java `int` and the `EntityRefType` is reduced to a reference to the class generated from the target entity. The `->$` is a so-called reference macro. It contains code (not shown) that "rewires" the reference (that points to the `Double` class *in the template code*) to a reference to the class generated from the target entity.

MPS templates work differently from normal text generation templates such as Xpand [5], Jet [6] or StringTemplate [7], since they are actually model-to-model transformations. However, as a consequence of MPS' language composition facilities, the concrete syntax of the target language is used in the template — it is projected like any other program. However, this means that the *template code itself* must be valid in terms of the target language.

Fig. 7 shows the `map_Entity` template. It generates a complete Java class from an input `Entity`. To understand how templates work in MPS we discuss in more detail the generation of Java fields for each `Entity Attribute`. Writing this template proceeds in the following way:

– Developers first write a structurally correct example code in the target language. To generate a field into a class for each `Attribute` of an `Entity`, one would first add a regular field to a class: `private int aField;` (as shown in Fig. 7).
– Then so called Macros are attached to those program elements from the example code that have to be replaced with data from the transformation input model as the transformation executes. In the `Attribute` example in Fig. 7 we first attach a `LOOP` macro to the whole field. It contains an expression `node.attributes;` where `node` refers to the input `Entity` (this code is entered in the Inspector window and is not shown in the screenshot).

[5] http://www.eclipse.org/modeling/m2t/?project=xpand
[6] http://www.eclipse.org/modeling/m2t/?project=jet
[7] http://www.stringtemplate.org/

This expression returns the set of `Attributes` from the current `Entity`, so the `LOOP` iterates over all attributes of the entity and creates a field for each of them.

– At this point, each created field would be *identical* to `private int aField;`, the example code to which we attached the `LOOP` macro. To make the generated field specific to the particular `Attribute` we iterate over, we use more macros. A `COPY_SRC` macro is used to transform the `type`. `COPY_SRC` copies the input node (the inspector specifies the current attribute's `type` as the input here) and applies reduction rules (those defined in Fig. 6) to map types from the `entities` language to Java types. We then use a property macro (the `$` sign around `aField`) to change the `name` property of the field we generate from the dummy value `aField` to the name of the attribute we currently transform.

So instead of mixing template code and target language code (and separating them with some kind of escape character) we use annotation attached to regular, valid target language code. These annotations can be attached to arbitrary program elemnts. This way, the target language code in templates *is always structurally correct*, but it can still be annotated to control the transformation. Annotations are a general MPS mechanism not specific to transformation templates and are discussed in Section 4.5.

```
[root template
 input Entity]

public class $[map_Entity] extends <none> implements <none>

  $LOOP$[private $COPY_SRC$[int] $[aField]; ]

  public map_Entity() {
    <no statements>
  }

  $LOOP$[public void $[setter]($COPY_SRC$[int] newValue) {
           <<placeholder>> pre-set : $[attr]
           this.aField = newValue;
         }]

  $LOOP$[public $COPY_SRC$[int] $[getter]() {
           return aField;
         }]
```

Fig. 7 The template for creating a Java class from an `Entity`. The generated class contains a field, a getter and a setter for each of the `Attributes` of the `Entity`. The running text explains the details.

4 Language Composition with MPS

In this section we discuss the four language and IDE modularization and compositiontechniques introduced in the previous section, plus an additional one that works only with a projectional editor such as MPS. For the first four, we provide a concise verbal definition plus a set of formulas to nail down the specifics. We then illustrate each technique with a detailed MPS-based example that relates to the `entities` language introduced earlier.

4.1 Language Referencing

Language referencing enables *homogeneous* fragments with cross-references among them, using *dependent* languages (Fig. 8).

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

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

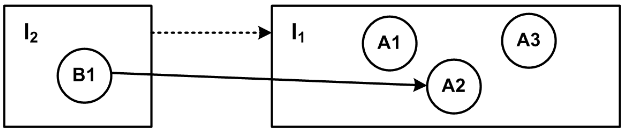

Fig. 8 Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, normal arrows = associations, hollow-triangle-arrow for inheritance.)

As an example, for language referencing we define a language `uispec` for defining user interface forms for `entities`. Below is an example. Note how the form is another, separate fragment. It is a *dependent* fragment, since it references elements from another fragment (expressed in the `entities` language). Both fragments are *homogeneous* since they consist of sentences expressed in a single language.

```
form CompanyStructure
  uses Department
  uses Employee
  field Name: textfield(30) -> Employee.name
  field Role: combobox(Boss, TeamMember) -> Employee.role
  field Freelancer: checkbox -> Employee.freelancer
  field Office: textfield(20) -> Department.description
```

■ *Structure and Syntax.* The abstract syntax for the `uispec` language is shown in Fig. 9. The `uispec` language extends[8] the `entities` language. This means that concepts from the `entities` language can be used in the definition of language concepts in the `uispec` language. A `Form` owns a number of `EntityReferences`, which in turn reference the `Entity` concept. Also, `Fields` refer to the `Attribute` that shall be edited via the field. Below is the definition of the `Field` concept. It owns a `Widget` and refers to an `Attribute`.

```
concept Field extends BaseConcept
  properties:
    label : string
  children:
    Widget widget 0..1
  references:
    Attribute attribute 0..1
```

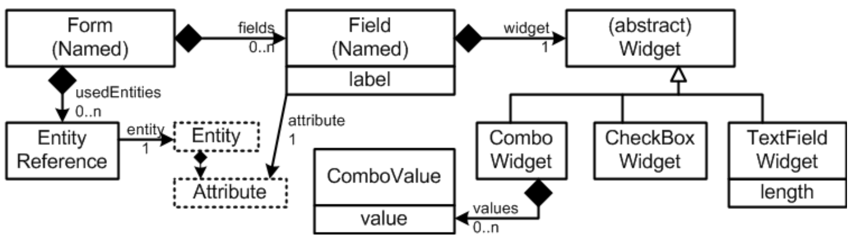

Fig. 9 The abstract syntax of the `uispec` language. Dotted boxes represent classes from another language (here: the `entities` language). A `Form` contains `EntityReferences` that connect to an `entities` model. A `Form` also contains `Fields`, each referencing an `Attribute` from an `Entity` and containing a `Widget`.

Note that there is no composition of concrete syntax, since the programs written in the two composed languages remain separated into their own fragments. No clashes between names of concepts may occur in this case.

■ *Type System.* There are limitations regarding which widget can be used with which attribute type. This typing rule is defined in the `uispec` language and references types from the `entities` language. The following is the code for the type check. We use a non-typesystem rule to illustrate how constraints can be written that do not use the inference engine introduced above.

[8] MPS uses the term "extension" whenever the definition of one language uses or refers to concepts defined in another language. This is not necessarily an example of language extension as defined in this paper.

```
checking rule checkTypes for Field as field {
    if (field.widget.isInstanceOf(CheckBoxWidget)
         && !(field.attribute.type.isInstanceOf(BooleanType))) {
      error "checkbox can only be used with booleans" -> field.widget;
    }
    if (field.widget.isInstanceOf(ComboWidget)
         && !(field.attribute.type.isInstanceOf(StringType))) {
      error "combobox can only be used with strings" -> field.widget;
} }
```

■ *Generation.* The defining characteristic of language referencing is that the two languages only reference each other, and the instance fragments are dependent, but homogeneous. No syntactic integration is necessary in this case. In this example, the generated code exhibits the same separation. From the form definition, we generate a Java class that uses Java Swing to build the form. It *uses* the beans generated from the `entities`. The classes are instantiated, and the setters are called. The generators are separate but they are *dependent*, they share information. Specifically, the forms generator knows about the names of the generated entity classes, as well as the names of the setters and getters. This is implemented by defining a couple of behaviour methods on the `Attribute` concept that are called from both generators (the colon represents the node cast operator and binds tightly; the code below casts the attribute's parent to `Entity` and then accesses the `name` property).

```
concept behavior Attribute {
  public string qname() {
    this.parent : Entity.name + "." + this.name;
  }
  public string setterName() {
    "set" + this.name.substring(0, 1).toUpperCase() + this.name.substring(1);
  }
  public string getterName() {
    "get" + this.name.substring(0, 1).toUpperCase() + this.name.substring(1);
  }
}
```

The original `entities` fragment is still *sufficient* for the transformation that generates the Java Bean. The form fragment is not sufficient for generating the user interface, because it needs the entity fragment. This is not surprising since *dependent* fragments can never be sufficient for a transformation: the transitive closure of all dependencies has to be made available.

4.2 Language Extension

Language extension enables *heterogeneous* fragments with *dependent* languages (Fig. 10). 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

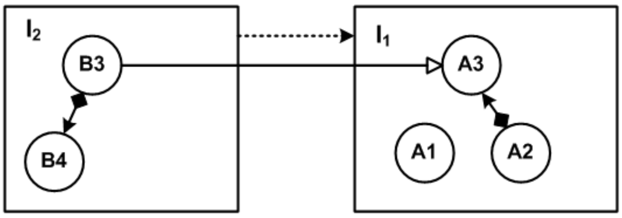

Fig. 10 Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

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:

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

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

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

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.

As an example, we extend the MPS base language with block expressions and placeholders. These concepts make writing generators *that generate base language code* much simpler. Fig. 11 shows an example. We use a screenshot instead of text because we use non-textual notations (the big brackets) and color.

■ *Structure and Syntax.* A block expression is a block that can be used where an `Expression` is expected [6]. The block can contain any number of statements; `yield` can be used to "return values" from within the block. So, in some sense, a block expression is an "inlined method", or a closure that is defined and called directly (an optional name property of a block expression is then used as the method name). The generator of the block expression transforms it into just this structure:

```
okButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent p0) {
    Employee aEmployee = new Employee();
    aEmployee.setName(retrieve_name(aEmployee, widget0));
  }
  public String retrieve_name(Employee aEmployee, JComponent widget0) {
    String newValue = ((JTextField) widget0).getText();
    return newValue;
  }
}
```

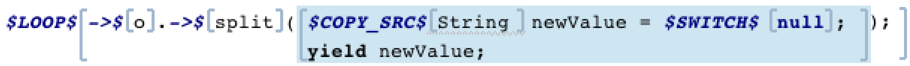

Fig. 11 Block Expressions (rendered with a shaded background) are basically anonymous inline methods. Upon transformation, an actual method is generated that contains the block content, and the block expression is replaced with a call to this generated method. Block expression are used mostly when implementing generators; this screenshot shows a generator that uses a block expression.

The `jetbrains.mps.baselanguage.exprblocks` language extends MPS' BaseLanguage. The block expression is used in places where the base language expects an `Expression`. This is why a `BlockExpression` extends `Expression`. Consequently, fragments that use the `exprblocks` language, can now use `BlockExpressions` in addition to the concepts provided by the base language. The fragments become *heterogeneous*, because the languages are mixed.

```
concept BlockExpression extends Expression implements INamedConcept
  children:
    StatementList body 1
```

■ *Type System.* The type of the `yield` statement is the type of the expression that is `yield`ed, specified by `typeof(yield) :==: typeof(yield.result);` (so the type of `yield 1;` would be `int`, because the type of `1` is `int`). Since the `BlockExpression` is used as an expression, it has to have a type as well. Since it is not explicitly specified, the type of the `BlockExpression` is the common super type of the types of all the yields. The following typing rule computes this type:

```
var resultType ;
for (node<BlockExpressionYield> y :
        blockExpr.descendants<BlockExpressionYield>) {
  resultType :==: typeof(y.result);
}
typeof(blockExpr) :==: resultType;
```

■ *Generator.* The generator for `BlockExpression`s reduces the new concept to pure base language: it performs assimilation. It transforms a heterogeneous fragment (using BaseLanguage and `exprblocks`) to a homogeneous fragment (using only BaseLanguage). The first step is the creation of the additional method for the block expression (Fig. 12).

```
concept     BlockExpression
inheritors  false
condition   <always>
    -->
    weave_BlockExpression
    context : (node, genContext, operationContext)->node< > {
                  node<ClassConcept> cls = node.ancestor<concept = ClassConcept, +>;
                  genContext.get copied output for (cls);
              }
```

Fig. 12 We use a weaving rule to create an additional method for a block expression. A weaving rule processes an input element (`BlockExpression`) by creating another node in a different place. The context function defines this other place. In this example, it simply gets the class in which we have defined the particular block expression, so the additional method is generated into that class.

The template shown in Fig. 13 shows the creation of the method. It assigns a mapping label to the created method. The mapping label creates a mapping between the `BlockExpression` and the created method. We will use this label to refer to this generated method when we generate the method call that replaces the `BlockExpression` (Fig. 14).

```
<TF b2M [public $COPY_SRC$[string] $[amethod]($LOOP$v2P[ $COPY_SRC$[int] $[a]]) {  ] TF> ]])
          $COPY_SRCL$[return "hallo"; ]
        }
```

Fig. 13 This generator template creates a method from the block expression. It uses `COPY_SRC` macros to replace the dummy `string` type with the computed return type of the block expression, inserts a computed name, adds a parameter for each referenced variable outside the block, and inserts all the statements from the block expression into the body of the method. The `b2M` (block-to-method) mapping label is used later when generating the call to this generated method (shown in Fig. 14).

A second concept introduced by the `exprblocks` language is the `PlaceholderStatement`. It extends `Statement` so it can be used inside method bodies. It is used to mark locations at which subsequent generators want to add additional code. These subsequent generators will use a reduction rule to replace the placeholder with whatever they want to put at this location. It is a means to building extensible generators, as we will see later.

```
public void caller() {
  int j = 0;
  <TF [ ->$[callee]($LOOP$[$COPY_SRC$[j]]) ] TF>;
}
```

Fig. 14 Here we generate the call to the previously generated method. We use the mapping label `b2M` to refer to the correct method (not shown; happens inside the reference macro). We pass in the environment variables as actual arguments using the `LOOP` and `COPY_SRC` macros.

In the classification section (Section 1.4) we mentioned that we consider restriction as a form of extension. To illustrate this point we restrict the block expression regarding the children it may have: we prevent the use of **`return`** statements inside block expressions (the reason for this restriction is that the way we generate from the block expressions cannot handle return statements). To achieve this, we add a **`can be ancestor`** constraint to the `BlockExpression`:

```
concept constraints for BlockExpression {
  can be ancestor:
    (operationContext, scope, node, childConcept, link)->boolean {
      childConcept != concept/ReturnStatement/;
}   }
```

The `childConcept` variable represents the concept of which an instance is about to be added under a `BloclExpression`. The constraint expression has to return `true` if the respective `childConcept` is valid in this location. We make sure the `childConcept` is not a `ReturnStatement`. Note how this constraint is written *from the perspective of the ancestor* (the `BlockExpression`). MPS also supports writing constraints from the perspective from the child. This is important to keep dependencies pointing in the right direction.

Extension comes in two flavors. One really feels like extension, and the other one feels more like embedding. We have described the one that feels like extension in this section: we provide (a little, local) additional syntax to an otherwise unchanged language (block expressions and placeholders). The programs still essentially look like Java programs, and in a few particular places, something is different. Extension with embedding flavor is where we create a completely new language, but reuse some of the syntax provided by a base language. For example, we could create a state machine language that reuses Java's expression and types in guard conditions. This use case *feels* like embedding (we embed syntax from the base language in our new language), but in the classification according to syntactic integration and dependencies, it is still extension. Embedding would prevent a dependency between the state machine language and Java.

4.3 Language Reuse

Language reuse (Fig. 15) enables *homogenous* fragments with *independent* languages. Given are two independent languages l_2 and l_1 and two fragment f_2 and f_1. f_2 depends on f_1, so that

$$\exists r \in Refs_{f_2} \mid fo(r.from) = f_2 \wedge (fo(r.to) = f_1 \vee fo(r.to) = f_2) \quad (11)$$

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

$$\exists r \in Refs_{l_A} \mid lo(r.from) = l_A \ \wedge \ lo(r.to) = l_1 \quad (12)$$

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.

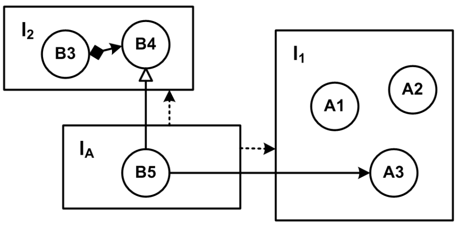

Fig. 15 Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in another fragment expressed with l_1. To do this, an adapter language l_A is added that depends on both l_1 and l_2, using inheritance and referencing to adapt l_1 to l_2.

Language reuse covers the case where a language has been developed independent of the context in which it should be reused. The respective fragments remain homogeneous. In this paper, we cover two alternative cases: the first case addresses a persistence mapping language. The generated code is separate from the code generated from the `entities` language. The second case described a language for role-based access control. The generated code has to be "woven into" the `entities` code to check permissions when setters are called.

4.3.1 Separated Generated Code

■ *Structure and Syntax.* relmapping is a reusable language for mapping arbitrary data to relational tables. The relmapping language supports the definition of relational table structures, but leaves the actual mapping to the source data unspecified. As you adapt the language to a specific reuse context, you have to specify this mapping. The following code shows the reusable part. A database is defined that contains tables with columns. Columns have (database-specific) data types.

```
Database CompanyDB
  table Departments
    number id
    char descr
  table People
    number id
    char name
    char role
    char isFreelancer
```

Fig. 16 shows the structure of the relmapping language. The abstract concept `ColumnMapper` serves as a hook: if we reuse this language in a different context, we extend this hook by context-specific code.

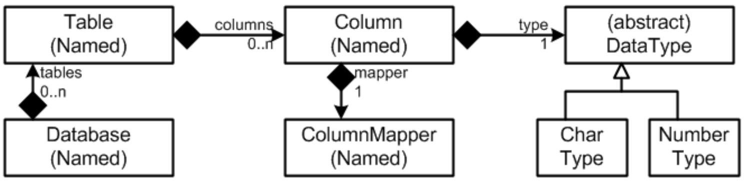

Fig. 16 A `Database` contains `Tables` which contain `Columns`. A column has a name and a type. A column also has a `ColumnMapper`. This is an abstract concept that determines where the column gets its data from. It is a hook intended to be specialized in sublanguages, specific to the particular reuse context.

The `relmapping_entities` language extends relmapping and adapts it for reuse with the `entities` language. To this end, it provides a subconcept of `ColumnMapper`, the `AttributeColMapper`, which references an `Attribute` from the `entities` language as a means of expressing the mapping from the attribute to the column. The column mapper is projected on the right of the field definition, resulting in the following (heterogeneous) code fragment:

```
Database CompanyDB
  table Departments
    number id <- Department.id
    char descr <- Department.description
  table People
    number id <- Employee.id
```

```
    char name <- Employee.name
    char role <- Employee.role
    char isFreelancer <- Employee.freelancer
```

■ *Type System.* The type of a column is the type of its `type` property. In addition, the type of the column must also conform to the type of the column mapper, so the concrete subtype must provide a type mapping as well. This "typing hook" is implemented as an abstract behaviour method `typeMappedToDB` on the `ColumnMapper`. It is acceptable from a dependency perspective to have this typing hook, since relmapping is designed to be extensible. The typing rules then look as follows:

```
typeof(column)       :==: typeof(column.type);
typeof(column.type)  :==: typeof(column.mapper);
typeof(columnMapper) :==: columnMapper.typeMappedToDB();
```

The `AttributeColMapping` concept from the `relmapping_entities` implements this method by mapping ints to numbers, and everything else to chars.

```
public node<> typeMappedToDB()
  overrides ColumnMapper.typeMappedToDB {
  node<> attrType = this.attribute.type.type;
  if (attrType.isInstanceOf(IntType)) { return new node<NumberType>(); }
  return new node<CharType>();
}
```

■ *Generator.* The generated code is also separated into a reusable part (a class generated by the generator of the relmapping language) and a context-specific subclass of that class, generated by the `relmapping_entities` language. The generic base class contains code for creating the tables and for storing data in those tables. It contains abstract methods that are used to access the data to be stored in the columns. The dependency structure of the generated fragments, as well as the dependencies of the respective generators, resembles the dependency structure of the languages: the generated fragments are dependent, and the generators are dependent as well (they share the name and implicitly the knowledge about the structure of the class generated by the reusable relmapping generator). A relmapping fragment (without the concrete column mappers) is sufficient for generating the generic base class.

```
public abstract class CompanyDBBaseAdapter {

  private void createTableDepartments() {
    // SQL to create the Departments table
  }

  private void createTablePeople() {
    // SQL to create the People table
  }
```

```
  public void storeDepartments(Object applicationData) {
    StringBuilder sql = new StringBuilder();
    sql.append("insert into" + "Departments" + "(");
    sql.append("" + "id");
    sql.append(", " + "descr");
    sql.append(") values (");
    sql.append("" + "\"" + getValueForDepartments_id(applicationData) + "\"");
    sql.append(", " + "\"" + getValueForDepartments_descr(applicationData) + "\"");
    sql.append(")");
  }

  public void storePeople(Object applicationData) {
    // like above
  }

  public abstract String getValueForDepartments_id(Object applicationData);

  public abstract String getValueForDepartments_descr(Object applicationData);

  // abstract getValue methods for the People table
}
```

The subclass generated by the generator in the `relmapping_entities` language implements the methods defined by the generic superclass. The interface, represented by the `applicationData` object, has to be kept generic so any kind of user data can be passed in. Note how this class references the beans generated from the `entities`. The generator for `entities` and the generator for `relmapping_entities` are dependent. The information shared between the two generator is the names of the classes generated from the `entities`. The code generated from the relmapping language is designed to be extended by code generated from a sublanguage (the abstract getValue methods). This is acceptable, since the relmapping language itself is intended to be extended to adapt it to a new reuse context.

```
public class CompanyDBAdapter extends CompanyDBBaseAdapter {
  public String getValueForDepartments_id(Object applicationData) {
    Object[] arr = (Object[]) applicationData;
    Department o = (Department) arr[0];
    String val = o.getId() + "";
    return val;
  }
  public String getValueForDepartments_descr(Object applicationData) {
    Object[] arr = (Object[]) applicationData;
    Department o = (Department) arr[0];
    String val = o.getDescription() + "";
    return val;
  }
}
```

4.3.2 Interwoven generated code

■ *Structure and Syntax.* `rbac` is a language for specifying role-based access control, to specify access permissions for the `entities`.

```
users: user mv : Markus Voelter
       user ag : Andreas Graf
       user ke : Kurt Ebert

roles: role admin : ke
       role consulting : ag, mv

permissions:   admin, W : Department
               consulting, R : Employee.name
```

The structure is shown in Fig. 17. Like relmapping, it provides a hook, in this case, `Resource`, to adapt it to context languages. The sublanguge `rbac_entities` provides two subconcepts of `Resource`, namely `AttributeResource` to reference to an attribute, and `EntityResource` to refer to an `Entity`, to define permissions for `entities` and their attributes.

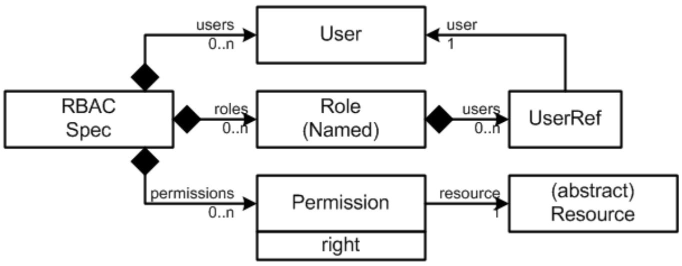

Fig. 17 Language structure of the `rbac` language. An `RBACSpec` contains `Users`, `Roles` and `Permissions`. Users can be members in several roles. A permission assigns a role and right (read, write) to a `Resource` (such as an `Entity` or an `Attribute`).

■ *Type System.* No type system rules apply here, because none of the concepts added by the `rbac` language are typed.

■ *Generator.* What distinguishes this case from the relmapping case is that the code generated from the `rbac_entities` language is *not* separated from the code generated from the `entities`. Instead, inside the setters of the Java beans, a permission check is required.

```
public void setName(String newValue) {
  // check permissions (from rbac_entities)
  if (new RbacSpecEntities().currentUserHasWritePermission("Employee.name")) {
    throw new RuntimeException("no permission");
  }
  this.name = newValue;
}
```

The generated fragment is homogeneous (it is all Java code), but it is *multi-sourced*, since several generators contribute to the same fragment. To implement this, several approaches are possible:

– We could use AspectJ (`http://www.eclipse.org/aspectj/`). This way, we could generate separate Java artifacts (all single-sourced) and then use the aspect weaver to ”mix” them. While this would be a simple approach in terms of MPS (because we only generate singled-sourced artifacts), it fails to illustrate advanced MPS generator concepts. So we don’t use this approach here.
– An interceptor (see the Interceptor pattern in [9]) framework could be added to the generated Java Beans, with the generated code contributing specific interceptors (effectively building a custom aspect oriented programming (AOP) solution). We will not use this approach either, for the same reason we don’t chose AspectJ.
– We could ”inject” additional code generation templates to the existing `entities` generator from the `rbac_entities` generator. This would make the generators *woven* as opposed to just dependent. However, weaving generators in MPS is not supported, so we cannot use this approach.
– We could define a hook in the generated Java beans code and then have the `rbac_entities` generator contribute code to this hook. This is the approach we will use. The generators remain dependent because they share the way the hook works.

Notice that only the AspectJ solution can work without any pre-planning from the perspective of the `entities` language, because it avoids mixing the generated code artifacts (it is handled ”magically” by AspectJ). All other solutions require the original `entities` generator to ”expect” certain extensions.

In our case, we have modified the original generator in the `entities` language to contain a `PlaceholderStatement` (Fig. 18). In every setter, the placeholder acts as a hook at which subsequent generators can add statements. While we have to pre-plan *that* we want to extend the generator in this place, we do not have to predefine *how*. The placeholder contains a key into the session object that points to the currently processed attribute. This way, the subsequent generator can know from which attribute the method with the placeholder in it was generated.

```
$LOOP$[public void $[setter]($COPY_SRC$[int] newValue) {
         <<placeholder>> pre-set : $[attr]
         this.aField = newValue;
       }                                                  ]
```

Fig. 18 This generator fragment creates a setter method for each attribute of an `Entity`. The `LOOP` iterates over all Attributes. The `$` macro computes the name of the method, and the `COPY_SRC` macro on the argument type computes the type. The placeholder is used later to insert the permission check.

The `rbac_entities` generator contains a reduction rule for `PlaceholderStatements`. So when the generator encounters a placeholder (that has been put there by the `entities` generator) it removes it and inserts the code that checks for the permission (Fig. 19). To make this work we have to specify in the generator priorities that this generator runs **strictly after** the **entities** generator (since the **entities** generator has to create the placeholder) and **striclty before** the BaseLanguage generator (which transforms BaseLanguage code into Java text for compilation). Priorities specify a partial ordering (cf. the **strictly before** and **strictly after**) on generators and can be set in the generator priorities dialog (not shown). Note that we specifying the priorities does not introduce additional language dependencies, modularity is retained.

```
reduction rules:
  concept     PlaceholderStatement
  inheritors false
  condition   (node, genContext, operationContext)->boolean {
                node.name.equals("pre-set");
              }
     --> content node:
           public void dummy() {
             <TF {{ // transparent block
                 // check permissions (from rbac_entities)
                 if (new ->$[RbacSpecEntities]().hasWritePermission("$[res]
                     ")) { throw new RuntimeException("no permission"); }
                 }} TF>
           }
```

Fig. 19 This reduction rule replaces `PlaceholderStatements` with a permission check. Using the condition, we only match those placeholders whose identifier is `pre-set` (notice how we have defined this identifier in the template shown in Fig. 18). The inserted code queries another generated class that contains the actual permission check. A runtime exception is thrown if the permission check fails.

4.4 Language Embedding

Language embedding enables *heterogeneous* fragments with *independent* languages (Fig. 15, 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

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

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

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

As an example we embed an existing, embeddable `expressions` language into the `uispec` language. To do this, we *do not* modify either the `uispec` language or the expression language, since, in case of embedding, none of them may have a dependency on the other. Here is an example program using the resulting language. Note the use of expressions behind the `validate` keyword:

```
form CompanyStructure
  uses Department
  uses Employee

  field Name: textfield(30) -> Employee.name validate lengthOf(Employee.name) < 30
  field Role: combobox(Boss, TeamMember) -> Employee.role
  field Freelancer: checkbox -> Employee.freelancer
        validate if (isSet(Employee.worksAt)) Employee.freelancer == true else
                    Employee.freelancer == false
  field Office: textfield(20) -> Department.description
```

■ *Structure and Syntax.* We create a new `uispec_validation` that extends `uispec` and it also extends `expressions`. Fig. 20 shows the structure. To be able to use the expressions, the user has to instantiate a `ValidatedField` instead of a `Field`. `ValidatedField` is also defined in `uispec_validation` and is a subconcept of `Field`.

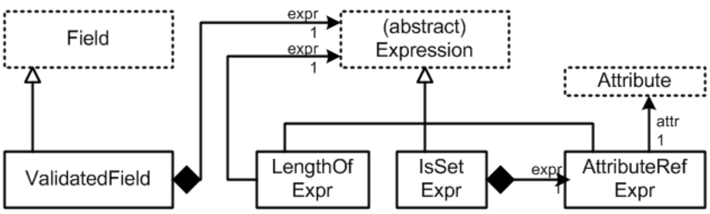

Fig. 20 The `uispec_validation` language defines a subtype of `uispec.Field` that contains an `Expression` from a reusable `expressions` language. The language also defines a couple of additional expressions, specifically the `AttributeRefExpr`, which can be used to refer to attributes of entities.

To support the migration of existing models that use `Field` instances, we provide an intention. An intention (known as a quick fix in Eclipse) is an in-place model transformation that can be triggered by the user by selecting it from the intentions menu accessible via `Alt-Enter`. This particular intention is defined for a `Field`, so the user can press `Alt-Enter` on a `Field` and select `Make Validated Field`. This transforms an existing `Field` into a `ValidatedField`, so that validation expressions can be entered. The core of the intention is the following script, which performs the actual transformation:

```
execute(editorContext, node)->void {
    node<ValidatedField> vf = new node<ValidatedField>();
    vf.widget = node.widget;
    vf.attribute = node.attribute;
    vf.label = node.label;
    node.replace with(vf);
}
```

As mentioned, the `uispec_validation` language extends the `uispec` and `expressions` languages. `ValidatedField` has a property `expr` that contains the actual `Expression`. As a consequence of polymorphism, we can use any existing subconcept of `Expression` defined in the `expressions` language here. So without doing anything else, we could write `20 + 40 > 10`, since integer literals and the `+` operator are defined as part of the embedded `expressions` language. However, to write anything useful, we have to be able to reference entity attributes from within expressions. To achieve this, we create the `AttributeRefExpr` as shown in Fig. 20. We also create `LengthOfExpr` and `IsSetExpression` as further examples of how to adapt an embedded language to its new context — i.e. the `uispec` and `entities` languages. The following is the structure definition of the `LengthOfExpr`.

```
concept LengthOfExpr extends Expression
  properties:
    alias = lengthOf
  children:
    Expression  expr  1
```

Note how it defines an `alias`. The `alias` is used to pick the concept from the code completion menu. If the user is in expression context, he must type the `alias` of a concept to pick it from the code completion menu. Typically, the alias is selected to match the leading keyword of the concept's projection. The `LengthOfExpr` is projected as `lengthOf(something)`, so by choosing the alias to also be `lengthOf`, the concept can be entered naturally.

The `AttributeRefExpr` references entity attributes. However, it may only reference those attributes of those entities that are used in the form within which we define the validation expression. The following is the code for the search scope:

```
(model, scope, referenceNode, linkTarget, enclosingNode)
                             ->join(ISearchScope | sequence<node< >>) {
  nlist<Attribute> res = new nlist<Attribute>;
  node<Form> form = enclosingNode.ancestor<Form, +>;
  for (node<EntityReference> er : form.usedEntities) {
    res.addAll(er.entity.attributes);
  }
  res;
}
```

Notice that the actual syntactic embedding of the expressions in the `uispec_validation` language is not a problem because of how projectional editors work. No ambiguities may arise. We simply define `Expression` to be a child of the `ValidatedField`.

■ *Type System.* The general challenge here is that primitive types such as `int` and `string` are defined in the `entities` language and in the reusable expression language. Although they have the same names, they are not the same types. The two sets of types must be mapped. Here are a couple of examples. The type of the `IsSetExpression` is by definition `expressions.BooleanType`. The type of the `LengthOf`, which takes an `AttrRefExpression` as its argument, is `expressions.IntType`. The type of an attribute reference is the type of the attribute's `type` property, as in `typeof(are) :==: typeof(are.attr.type);`. However, consider now the following code:

```
field Freelancer: checkbox -> Employee.freelancer
      validate if (isSet(Employee.worksAt)) Employee.freelancer == true else
                  Employee.freelancer == false
```

This code states that if the `worksAt` attribute of an employee is set, then its `freelancer` attribute must be `true` else it must be `false`. It uses the equals operator from the expressions language. However, that operator expects two `expressions.BooleanType` arguments, but the type of the `Employee.freelancer` is `entities.BooleanType`. In effect, we have to override the typing rules for the expressions language's equals operator. Here is how we do it, using `Equals` as an example.

In the expressions language, we define overloaded operation rules. We specify the resulting type for an `EqualsExpression` depending on its argument types. Below is the code in the expressions language that defines the resulting type to be `boolean` if the two arguments are `Equallable`:

```
operation concepts: EqualsExpression
  left operand type: new node<Equallable>()
  right operand type: new node<Equallable>()
operation type: (operation, leftOperandType, rightOperandType)->node< > {
  <boolean>;
}
```

In addition to this code, we have to specify that `expressions.BooleanType` is a subtype of `Equallable`, so this rule applies if we use equals with two `expressions.BooleanType` arguments. We have to tie this overloaded operation specification into a regular type inference rule.

```
rule typeof_BinaryExpression for BinaryExpression as binaryExpression {
  node<> opType = operation type( binaryExpression , left , right );
  if (opType != null) {
    typeof(binaryExpression) :==: opType;
  } else {
    error "operator " + binaryExpression.concept.name +
          " cannot be applied to these operand types " +
```

```
        left.concept.name + "/" + right.concept.name
            -> binaryExpression;
  }
}
```

To override these typing rules to work with `entities.BooleanType`, we simply provider another overloaded operation specification in the `uispec_validation` language:

```
operation concepts: EqualsExpression
  one operand type: <boolean> // this is the entities.BooleanType!
operation type: (operation, leftOperandType, rightOperandType)->node< > {
  <boolean>;  // this is the expressions.BooleanType
}
```

■ *Generator.* The generator has to create BaseLanguage code, which is then subsequently transformed into Java text. To deal with the transformation of the expressions language, we can do one of two things:

– We can use the expression's language existing to-text generator and wrap the expressions in some kind of `TextHolderStatement`. Remember that we cannot simply embed text in BaseLanguage, since that would not work structurally. A wrapper is necessary.
– Alternatively, we can write a (reusable) transformation from expressions code to BaseLanguage code; these rules would be used as part of the transformation of `uispec` and `uispec_validation` code to BaseLanguage.

Since many DSLs will map code to BaseLangauge, it is worth the effort to write a reusable generator from `uispec_validation` expressions to BaseLanguage expressions. We choose this second alternative. The generated Java code is multi-sourced, since it is generated by two independent code generators.

Expression constructs from the reusable expr language and those of BaseLanguage are almost identical, so this generator is trivial. We create a new language project `de.voelter.mps.expressions.blgen` and add reduction rules. Fig. 21 shows some of these reduction rules.

In addition to these, we also need reduction rules for those new expressions that we have added specifically in the `uispec_validation` language (`AttrRefExpression, isSetExpression, LengthOf`). Those are defined in `uispec_validation`. As an example, Fig. 22 shows the rule for handling the `AttrRefExpression`. The validation code itself is "injected" into the UI form via the same placeholder reduction as in the case of the `rbac_entities` language.

Just as in the discussion on extension (Section 4.2), we may want to use constrains to restrict the embedded language in the context of a validation rule. Imagine we wouldn't embed a small, reusable expression language, but the expressions part of C. It defines all kinds of operators relating to pointers, bit shifting and other C-specific things that are not relevant in the validation

```
reduction rules:
 [concept    MultiExpression] --> <T  $COPY_SRC$[1] * $COPY_SRC$[2]  T>
 [inheritors false          ]
 [condition  <always>       ]

 [concept    FalseLiteral] --> <T  false  T>
 [inheritors false       ]
 [condition  <always>    ]

 [concept    BooleanType] --> <T  boolean  T>
 [inheritors false      ]
 [condition  <always>   ]

 [concept    IfExpression] --> <T  $COPY_SRC$[true] ?
 [inheritors false       ]          $COPY_SRC$[true] : $COPY_SRC$[true]  T>
 [condition  <always>    ]
```

Fig. 21 A number of reduction rules that map the reusable expression language to BaseLanguage (Java). Since the languages are very similar, the mapping is trivial. For example, a `PlusExpression` is mapped to a `+` in Java, the left and right arguments are reduced recursively through the `COPY_SRC` macro.

```
reduction rules:
 [concept    AttributeRefExpr] --> content node:
 [inheritors false           ]     public void dummy( ) {
 [condition  <always>        ]       Object anObj = null;
                                     <TF [ ->$[anObj].->$[toString]( ) ] TF>;
                                   }
```

Fig. 22 References to entity attributes are mapped to a call to their getter method. The template fragment (inside the `<TF .. TF>`) uses two reference macros (->$) to "rewire" the reference to the Java bean instance, and the dummy `toString` method call to a call to the getter.

of UI fields. In this case we may want to use a `can be ancestor` constraint to restrict the use of those operators in the validation expressions.

As a consequence of MPS' projectional editor, no ambiguities may arise if multiple independent languages are embedded. Let us consider the potential cases for ambiguity:

Same Concpet Name: Embedded languages may define concepts with the same name as the host language. This will not lead to ambiguity because concepts have a unqiue ID as well. A program element will use this ID to refer to the concept whose instance it represents.

Same Concrete Syntax: The projected representation of a concept is not relevant to the functioning of the editor. The program would still be unambiguous to MPS even if *all elements had the same notation*. Of course it would be confusing to the users.

Same Alias: If two concepts that are valid at the same location use the same alias, then, as the user types the alias, it is not clear which of the two concepts should be instantiated. This problem is solved by MPS opening the code completion window and requiring the user to explicitly select which alternative to choose. Once the user has made the decision, the unique ID is used to create an unambiguous program tree.

4.5 Language Annotations

■ *Structure and Syntax.* In a projectional editor, the concrete syntax of a program is projected from the abstract syntax tree. A projectional system always goes from abstract to concrete, never from concrete to abstract (as parsers do). This has the important consequence that the concrete syntax does not have to contain all the data necessary to build the abstract syntax tree (which in case of parsers, is necessary). This has two consequences:

– A projection may be *partial* in the sense that the AST contains data that is not shown in the program. The information may, for example, only be changeable via intentions (discussed in Section 4.4), or the projection rule may project some parts of the program only in some cases, controlled by some kind of configuration data.
– It is also possible to project *additional* concrete syntax that is not part of the concrete syntax definition of the original language. Since the concrete syntax is never used as the information source, such additional syntax does not confuse the tool (in a parser-based tool the grammar would have to be changed to take into account this additional syntax to not derail the parser).

In this section we discuss the second alternative since it constitutes a form of language composition: the additional syntax is composed with the original syntax defined for the language. The mechanism MPS uses for this is called annotations. We have seen annotations when we discussed templates: an annotation is something that can be attached to arbitrary program elements and can be shown together with concrete syntax of the annotated element. In this section we use this approach to implement an alternative approach for the entity-to-database mapping. Using this approach, we can store the mapping from entity attributes to database columns directly in the `Entity`, resulting in the following code:

```
module company
  entity Employee {
    id : int -> People.id
    name : string -> People.name
    role : string -> People.role
    worksAt : Department -> People.departmentID
    freelancer : boolean -> People.isFreelancer
  }

  entity Department {
    id : int -> Departments.id
    description : string -> Departments.descr
  }
```

This is a heterogeneous fragment, consisting of code from the `entities`, as well as the annotations (e.g. `-> People.id`). From a concrete syntax perspective, the column mapping is "embedded" in the entity description. In the underlying persistent data structure, the mapping information is also actually stored in the entity model. However, the definition of the `entities` language does not know that this additional information is stored and projected "inside" entities. No modification to the `entities` language is necessary. Instead, we define an additional language `relmapping_annotations` which extends the `entities` language as well as the `relmapping` language. In this language we define the following concept:

```
concept AttrToColMapping extends NodeAnnotation
  references:
    Column column 1
  properties:
    role = colMapping
  concept links:
    annotated = Attribute
```

Node how the `AttrToColMapping` concept extends `NodeAnnotation`, a special concept predefined by MPS. Concept that extend `NodeAnnotation` have to provide a `role` property and an `annotated` concept link. Structurally, an annotation is a child of the node it annotates. So the `Attribute` has a new child of type `AttrToColMapping`, and the reference that contains the child is called `@colMapping` — the value of the `role` property. The `annotated` concept link points to the concept *to which this annotation can be added.* `AttrToColMapping`s can be annotated to intances of `Attribute`.

While structurally the annotation is a child of the annotated node, in the editor the relationship is reversed: The editor for `AttrToColMapping` wraps the editor for `Attribute`, as Fig. 23 shows. Since the annotation is not part of the original language, it must be attached to nodes via an intention.

```
editor for concept AttrToColMapping
  node cell layout:
    [- [> attributed node <] -> ( % column % -> * R/O model access * ) -]
```

Fig. 23 The editor for the `AttrToColMapping` embeds the editor of the concept it is annotated to (using the `attributed node` cell). It then projects the reference to the referenced column. This way the editor of the annotation has control of if and how the editor annotated element is projected.

Note that it is also possible to define the annotation source to be `BaseConcept`, which means the annotation can be attached to *any* node. The language that contains the annotation then has no dependency to any other language. This is useful for generic "metadata" such as documentation, requirements traces or presence conditions in product line engineering. We have described this in [49] and [47].

■ *Type System.* The same typing rules are necessary as in the `relmapping_entities` language described previously. They reside in `relmapping_annotations`.

■ *Generator.* The generator is also broadly similar to the previous example with `relmapping_entities`. It takes the `entities` model as the input, and then uses the column mappings in the annotations to create the entity-to-database mapping code.

5 Discussion

5.1 Limitations

The paper has drawn a very positive picture about the capabilities of MPS regarding language and IDE modularization and composition. However, there are some limitations and shortcomings in the system. Most of them are not conceptual problems, but just missing features. However, it is clearly evident that MPS has been developed over a long time, different problems have been solved in different ways, as the problem arose, instead of implementing a consistent, unified approach. I propose such an approach in Section 5.3.

■ *Syntax.* The previous examples show that meaningful language and IDE modularization and compositionis possible with MPS. Specifically, reuse and embedding of languages is possible. The challenge of grammar composition is not an issue in MPS, since no grammars and parsers are used. The fact that we hardly ever discuss syntactic issues in the above discussions is testament to this. Potential ambiguities are resolved by the user as he enter the program (discussed at the end of Section 4.4) — once entered, a program is always unambiguous. The luxury of not running into syntactic composition issues comes at the price of the projectional editor and the XML-based storage (we have discussed the drawbacks of projectional editors in Section 2). One particular shortcoming of MPS is that it is not possible to override the projection rule of a concept in a sublanguage (this feature is on the roadmap for MPS 3.0). If this were possible, ambiguities *for the user* in terms of the concrete syntax could be solved by changing the notation (or color or font) of existing concepts if they are used together with a particular other language.

■ *IDE.* This paper emphasizes IDE composition in addition to language composition. Regarding syntax highlighting, code completion, error marks on the program and intentions, all the composition approaches *automatically* compose those IDE aspects. No additional work is necessary by the languge developer. However, there are additional concerns an IDE may address including version control ingegration, profiling and debugging. Regarding version control integration, MPS provides diff/merge for most of today's version control systems on the level of the projected syntax — including for heterogeneous fragments. No support for profiling is provided. MPS comes with a debug framework that lets language developers create debuggers for their languages. However, this framework is relatively low-level and does not provide specific support for language composition and heterogeneous fragments. However, as part of the mbeddr project that develops an extensible version of the C programming language ([48]) we have developed a framework for extensible C debuggers. Developers of C extensions can easily define the how the extension integrates into the C debugger so that debugging on the *syntax of the extension* becomes possible for heterogeneous fragments. Visser et al. also describe debuggers for DSLs in [28].

■ *Evolution.* Composing languages leads to coupling. In the case of referencing and extension the coupling is direct, in the case of reuse and embedding the coupling is indirect via the adapter language. As a consequence of a change of the referenced/base/context/host language, the referencing/extension/reused/embedded language may have to change as well. MPS, at this time, provides no automatic way of versioning and migrating languages, so co-evolution has to be performed manually. In particular, a process discipline must be established in which dependent languages are migrated to new versions of a changed language.

■ *Type System.* Regular typing rules cannot be overridden in a sublanguage. Only the overloaded operations container can be overloaded (as their name suggests) from a sublanguage. As a consequence it requires some thought when designing a language to make the type system extensible in meaningful ways.

■ *Generators.* In the case of generators, language designers have to specify a partial ordering of mapping configurations using priorities. It is not easily possible to "override" an existing generator, but generators can run *before* existing ones. Generator extension is not possible directly. This is why we use the placeholders that are put in by earlier generators to be reduced by later ones. Obviously, this requires preplanning on the part of the developer of the generator that adds the placeholder.

5.2 Real-World use of MPS

The examples in this paper are toy examples — the simplest possible languages to illustrate the composition approaches. However, MPS scales to realistically sized systems, both in terms of language complexity and in terms of program size. The composition techniques — especially those involve syntactic composition — are used in practice. We illustrate this with two examples: embedded software and web applications.

■ *Embedded Software.* Embedded systems are becoming more software intensive and the software becomes bigger and more complex. Traditional embedded system development approaches use a variety of tools for various aspects of the system, making tool integration a major challenge. Some of the specific problems of embedded software development include the limited capability for meaningful abstraction in C, some of C's "dangerous" features (leading to various coding conventions such as Misra-C [17]), the proprietary and closed nature of modeling tools, the integration of models and code, traceability to requirements, long build times as well as management of product line variability. The mbeddr project ([9]) addresses the challenges with a different approach: incremental, modular extension of C with domain-specific language concepts.

mbeddr uses extension to add interfaces and components, state machines, measurement units to C. mbeddr is based on MPS, so users of mbeddr can build their own extensions. mbeddr implements all of C in less tha 10.000 lines of language implementation code. Scalability tests have shown that the system scales to at least 100.000 lines of equivalent C code.

A detailed description, including more details on language and program sizes and implementation effort can be found in [48].

■ *Web Development.* JetBrains' YouTrack issue tracking system is an interactive web application with many UI features known from desktop applications. YouTrack is developed completely with MPS and comprises thousands of Java classes, web page templates and other artifacts. The effort for building the necessary MPS-based languages will be repaid by future applications that build on the same web platform architecture and hence use the same set of languages. Language extension and composition is used to provide an integrated web development environment[10].

For example, the `dnq` language extends Java class definitions with all the information necessary to persist instances in a database via an object-relational mapper. This includes real associations (specifying navigability and composition vs. reference) or length specifications for string properties. Developers can access these associations just like regular fields. *dnq* also includes a collections language which supports the manipulation of collections in a way similar to .NET's Linq [32]. Other languages include *webr*, a language used for implementing interactions bwetwen the web page and the backend. It supports a unified programming model for application logic on the server and on the browser client. *webr* also provides first-class support for controllers. For example, controllers can declare actions and attach them directly to events of UI components. *webr* is well-integrated with *dnq*, so for example, it is possible to use a persistent entity as a parameter to a page. The database transaction is automatically managed during request processing.

In email communication with the author, JetBrains reported significant improvements in developer productivity for web applications. In particular, the time for new team members to become productive on the Youtrack team is reported to have been reduced from months to a few weeks, mostly because

[9] http://mbeddr.com
[10] Some details can be found in http://www.jetbrains.com/mps/docs/MPS_YouTrack_case_study.pdf

of the very tight integration in a single language of the various aspect of web application development.

5.3 A unified approach

Looking at the *different* mechanisms for extending and overriding structure, syntax, type system and generators, it is clear that a consistent approach for addressing all these aspects would be useful. I suggest to model the approach based on the principles of component-based design ([45]). All language aspects use components as the core structural building block. Components have types. The type of the component determines the kinds of facets it has. A facet is a kind of interface that exposes the (externally visible) ingredients of the component. The kinds of ingredients depend on the component type: a component of type *structure* exposes language concepts. A component of type *editor* exposes editors, type *type system* exposes type system rules, and so on. Each component type would use a different DSL for implementing the constructs it exposes. Here is the important point: a component (in a sublanguage) can specify an *advises* relationship to another component (from a super language). Then each of the facets can determine which facets from the advised component it wants to *preempt*, *enhance* or *override*:

- *preemption* means that the respective behavior is contributed before the behavior from the base language. A generator may use this to reduce a construct before the original generator gets a chance to reduce the construct.
- *enhancement* means that the sublanguage component is executed after the advised component from the base language. Notice that for declarative aspects where ordering is irrelevant, preempt and enhance are exchangable.
- *overriding* means that the original facet is completely shadowed by the new one. For example, this could be used to define a new editor for an existing construct.

This approach would provide the *same* way of packaging behavior for all language aspects, as well as a *single* way of changing that behavior in a sublanguage. To control the granularity at which preemption, enhancement or overriding is performed, the base language designer would have to group the structures or behaviors into suitably cut facets. This amount of preplanning is acceptable: it is just as in object-oriented programming, where behavior that should be overridable has to be packaged into its own method.

The approach could be taken further. Components could be marked as *abstract*, and define a set of parameters for which values need to be provided by non-abstract subcomponents. A language is abstract as long as it has at least one abstract component, for which no concrete subcomponent is provided. Component parameters could even be usable in structure definitions, for example as the base concept; this would make a language parametrizable regarding the base language it extends from.

6 Related Work

This paper addresses language modularity with MPS, a topic that concerns many different aspects. In this section we discuss related work focusing on modular parsers, projectional editing, modular compilers, projectional editing and modular IDEs. We conclude with a section on related work that does not fit these four categories.

6.1 Modular Parsers

As we have seen in this paper, modular composition of concrete syntax is the basis for many interesting approaches to language composition. Hence we start by discussing modularization and composition of grammars.

Kats, Visser and Wachsmut in [24] describe nicely the problems with non-declarative grammar specifications and the resulting problems for composition of independently developed grammars. The biggest problem with grammar formalisms that cover only subsets of the class of context-free grammars is that these are not closed under composition: resulting grammars are likely to fall out of the respective grammar class. Composition (without invasive change) is prohibited. Grammar formalisms that implement the full set of context-free grammars do not have this problem and support composition much better. Schwerdtfeger and van Wyk discuss discuss the issues surrounding grammar composition as well; they also describe a way of verifying early (i.e. before the actual composition attempt) if two grammars are composable or not [43].

For example, the Syntax Definition Formalism (SDF, [19]) is a scannerless GLR parser. Since it parses tokens and characters in a context-aware fashion, there will no ambiguities if grammars are composed that both define the same token or production *in different contexts*. This allows, for example, to embed SQL into Java (as Bravenboer et al. discuss in [4]). However, if the same syntactic form is used by the composed grammars *in the same location*, then some kind of disambiguation is necessary. Such disambiguations are typically called quotations and antiquotations and are defined in a third the grammar that describes the composition of two other independent grammars (discussed in [7]). The SILVER/COPPER system described by van Wyk in [50] solves the ambiguities via disambiguation functions written specifically for each combination of ambiguosly composed grammars. Note that in MPS such disambiguation is never necessary. We discuss the potential for ambiguity and the way solves the problem at the end of Section 4.4.

Given a set of extensions for a language, SILVER/COPPER allows users to include a subset of these extensions into a program as needed (this has been implemented for Java (AbleJ [52]) and and for the SPIN's Promela language (AbleP [30]). A similar approach is discussed for an SDF-based syste, in [8]. However, this only works as long as the set of included extensions (which have presumably been developed independent from each other) are not ambiguous with each other. In case of ambiguities, disambiguations have to be defined as described above.

Polyglot, an extensible compiler framework for Java [37] also uses an extensible formalism and parser to supports adding, modifying or removing productions and symbols defined in a base grammar. However, since Polyglot uses LALR grammars, users must make sure *manually* that the base language and the extension stays in the LALR subclass.

6.2 Projectional Editing

Projectional editing is an alternative approach for handling the relationship between the concrete syntax and the abstract syntax, i.e. it is an alternative to parsing. As we have seen, it simplifies modularization and composition.

Projectional editing (also known as structural editing) is not a new idea. An early example is the Incremental Programming Environment (IPE, [31]). It uses a structure editor for users to interact wit the program and then incrementally compiles and executes the resulting program tree. It supports the definition of several notations for the same program as well as partial projections. However, the projectional editor forces users to build the program tree top-down. For examplec, to enter `2 + 3` users first have to enter the `+` and then fill in the two arguments. This is very tedious and forces users to understand the program structure. MPS in contrast goes a long way in supporting editing gestures that much more resemble text editing. The IPE also does not address language modularity. In fact it comes with a fixed, C-like language. The IPE does not come with a facility to easily define new languages. It is not bootstrapped. Another projectional system is GANDALF [36]. Its ALOEGEN component generates projecitonal editors from a language specification. It has the same usability problems as described for IPE. This is nicely expressed in [39]: *Program editing time will be considerably slower than normal keyboard entry although actual time spent programming non-trivial programs should be reduced due to reduced error rates.*.

The synthesizer generator described in [42] also supports structural editing (also referred as template-based editing). However, at the fine-grained expression level, textual input and parsing is used. This removes many of the advantages of projectional editing in the first place, because simple language composition *even at the expression level* is prohibited. MPS does not use this “trick”, and instead supports projectional editing also on expression level.

Bagert and Friesen describe a multi-language syntax directed editor in [3]. However, this tool supports only referencing, syntactic composition is not supported.

In terms of contemporary projectional editors, The Intentional Domain Workbench (IDW) [44] is another projectional editor that has been used in real projects. An impressive presentation about its capabilities can be found in an InfoQ presentation titled ”Domain Expert DSL” (`http://bit.ly/10BsWa`).

IDW is conceptually very similar to MPS, although quite different in many details.

6.3 Modular Compilers

Modular compilers make use of modular parsers and add modular specification of semantics, including static semantics (constraints and type systems) as well as execution semantics.

Most systems describe static semantics using attribute grammars. Attribute grammars associate attributes with AST elements. These attributes can capture arbitrary data about the element (such as its type). Attributes of one element can be computed from attributes of related elements (such as children). Example of systems that make use of attribute grammars for type computation and type checking include SILVER (mentioned above) JastAdd [18] and LISA (discussed in more detail in the next section). Forwarding (introduced in [51]) is a mechanism that improves the modularity of attributed grammars by delegating the lookup of an attribute value to another element. MPS' type system is different from attribute grammars. Attributes values are calculated (recursively) from attributes of other, references nodes. MPS' type system rules are declarative: users specify typing rules for language concepts and MPS then "instantiates" each rule for each AST node. A solver then solves all type equations in that AST. This way, the typing rules of elements contributed by language extensions can *implicitly* affect the overall typing of the program.

For language extension, the execution semantics is defined in MPS via transformation down to the base language. In [50], van Wyk discusses under which circumstances such transformations are valid. In particular, the changes to the overall AST must be local. No global changes to the AST are allowed. This is to avoid unintended interactions between several independently developed extensions used in the same program. In MPS such purely local changes are called reduction rules. In our experience, it is also feasible to add additional elements to the AST *in select places* based on the use of an extension. In MPS, this is achieved using weaving rules. However, in both cases (local reduction and selective adding) there is no way to detect in advance whether using two extensions in the same program will lead to conflicts.

More formal ways to define semantics include denotational semantics, operational semantics and and a mapping to a formally defined action language. These have been modularized to make them composable. For example, Mosses describes modular structural operational semantics [35] and language composition by combining action semantics modules [11].

Aspect orientation supports the modularization of cross-cutting concerns. This has also been applied to language development. For example, Rebernak el al. [40] discuss AspectLisa and AspectG. AspectLisa supports adding new, cross-cutting attribute grammar attributes into a Lisa language definition. AspectG allows weaving additional action code into ANTLR grammars. Note that both AspectLisa and AspectG address semantics and not the grammar itself. They do not support aspect-oriented extension of the concrete syntax.

6.4 Modular IDEs

Now that we have discussed many of the fundamentals that enable modular languages, we can take at tools that, from a language definition, create a language aware-editor and other IDE aspects.

Among the early examples are the Synthesizer Generator [42], mentioned above, as well as the Meta Environment [25]. The provides an editor for languages defined via and ASF+SDF, i.e. it is parser-based. More recent tools in the ASF+SDF family include Rascal [26] and Spoofax [23]. Both provide Eclipse-based IDE support for languages defined via SDF. In both cases the IDE support for the composed languages is still limited (for example, at the time of this writing, Spoofax only provides syntax highlighting for an embedded language, but no code completion), but will be improved. For implementing semantics, Rascal uses a Java-like language that has been extended with features for program construction, transformation and analyses. Spoofax uses term rewriting based on the Stratego [5] language. An interesting tool is SugarJ [13] also based on SDF introduces SugarJ, which supports library based languages extension. [12] adds Spoofax-based IDE support.

SmartTools [1] supports generating editors for XML schemas. Based on assigning UI components to AS elements, it can project an editor for programs. However, this projectional editor does not try to emulate text-like editing as MPS does, so there is no convenient way for editing expressions. To do this, a grammar-based concrete syntax can be associated with a the AS elements defined in the schema. Based on this definition, SmartTools then provides a text-based representation for the language. However, this prevents syntax composition (as in used in extension and embedding). SmartTools only supports homogeneous files. Different UI components and grammars can be defined for the same AS, supporting multi-notation editing. Static semantics is implemented based on the Visitor pattern [16]. SmartTools provides support for much of the infrastructure and makes using Visitors simple. For transformation, SmartTools provides Xpp, a transformation language that provides a shorter syntax for XSLT-based XML transformations.

LISA [34] (mentioned earlier) supports the definition of language syntax (in a BNF-like way) and semantics (via attribute grammars) in one integrated specification language. It then then derives, among other things, a syntax-aware text editor for the language, as well as various graphical and structural viewing and editing facilities. Users can use inheritance and aspect-orientation to define sub-grammars. However, users have to make sure manually that those sub-grammars remain unambiguous. Combination of independently developed grammars (or sub-grammars) is not supported. LISA supports interactive debugging and program state visualization based on interpreting programs based on the semantic parts of the language specification.

Eclipse Xtext ([11]) generates sophisticated parser-based editors from an EBNF-like language specification. Syntactic composition is limited since Xtext is based on antlr [38] which is a classical two phase LL(*) parser. It is possible to make a language extend *one* other language. Concepts from the base language can be used in the sub language and it is possible to redefine grammar rules defined in the base language. Combination of independently defined extensions or embedding is not supported. Xtext's abstract syntax is based on EMF Ecore [12], it can be used together with any EMF-based model transformation and code generation tool (examples include Xpand, ATL, and Acceleo, all at `http://eclipse.org/modeling`). Static semantics is based on constraints written in Java or on frameworks that support declarative description of type systems such as XTS[13].

Monticore (`http://monticore.org`) is another parser-based language engineering environment that generates parsers, meta-models, and editors based on extended grammar. Currently, the group at RWTH Aachen university works on modularizing languages [27]. Languages can extend each other and can be embedded within each other. An important idea is the ability to not regenerate the parsers or any of the related tools after a combined language has been defined. However, ambiguities have to be avoided manually.

The Helvetia system by [41] supports language embedding and extension of Smalltalk using *homogeneous* extension, which means that the host language (Smalltalk) is also used for *defining* the extensions. The authors argue that the approach is independent of the host language and could be used with other host languages as well. While this is true in principle, the implementation strategy heavily relies on some aspects of the Smalltalk system that are not present for other languages. Also, since extensions are defined in the host language, the complete implementation would have to be redone if the approach were used with another language. This is particularly true for IDE support, where the Smalltalk IDE is extended using this IDE's APIs. MPS uses a *heterogeneous* approach which does not have these limitations: MPS provides a language-agnostic framework for language and IDE extension that can be used with any language, once the language is implemented in MPS.

Cedalion [**?**] is a host language for defining internal DSLs. It uses a projectional editor and semantics based on logic programming. Both Cedalion and MPS aim at combining the best of both worlds from internal DSLs (combination and extension of languages, integration with a host language) and external DSLs (static validation, IDE support, flexible syntax). Cedalion starts out from internal DSLs and adds static validation and projectional editing, the latter avoiding ambiguities resulting from combined syntaxes. MPS starts from external DSLs and add modularization, and, as a consequence of implementing base languages with the same tool, optional tight integration with general purpose host languages.

---

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

[12] http://eclipse.org/emf

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

For a general overview of language workbenches, please refer to the Language Workbench Competition at `http://languageworkbenches.net`. Participating tools have to implement a standardized language and document the implementation strategy. This serves as a good tutorial of the tool and makes them comparable. As of June 2012, the site contains 15 submissions.

6.5 Other Related Work

We already discussed the language modularization and composition approaches proposed by Mernik et. al. [33] in Section **??**. In the Helvetia paper [41] Renggli and his colleagues introduce three different flavors of language extension. A *pidgin* creatively bends the existing syntax of the host language to to extend its semantics. A *creole* introduces completely new syntax and custom transformations back to the host language. An *argot* reinterprets the semantics of valid host language code. In terms of this classification, both extension and embedding are creoles.

The idea of 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 [22]. Sergey Dmitriev discusses the idea of language and IDE extension in his article on Language Oriented Programming [10], 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 [2] where all macros have to begin with names, and a limited set of syntactic shapes is supported. In OpenJava [46], 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 language 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 [14]. Cascading is referred to as Piggybacking and Pipelining in Mernik et al.'s classification [33].

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 [20] 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. The landmark work of Hudak [21] introduces embedded DSLs as language extensions of Haskell. While Haskell provides advanced concepts that enable such extensions, the new DSLs are essentially just libraries built with the host language and are not first class language entities: they do not define their own syntax, compiler errors are expressed in terms of the host language, no custom semantic analyses are supported and no specific IDE-support is provided. Essentially all internal DSLs expressed with dynamic languages such as Ruby or Groovy, but also those embedded in static languages such as Scala suffer from these limitations.

7 Summary

MPS is powerful environment for language engineering. While not all of its features are unique (see Section 6), the referencing of flexible composition and the notational freedom as a consequence of the projectional approach is certainly convincing. I also want to emphasize that the tool also scales to realistic program sizes, the editor is very usable, and it integrates well with existing VCS (diff and merge is provided on the level of the concrete syntax). At the very minimum, the tool is a perfect environment for language experimentation in the context of academic and industrial research.

The major drawback of MPS is its non-trivial learning curve. Because it works so differently than traditional language engineering environments, and because it addresses so many aspects of languages (incl. type systems, data flow and refactorings) mastering the tools takes a significant investment in terms of time. I hope that in the future this investment will be reduced by better documentation and better defaults, to keep simple things simple and complex things tractable. There are initial ideas on how this could be done.

References

1. I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, and C. Pasquier. SmartTools: A Generator of Interactive Environments Tools. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2027 of *Lecture Notes in Computer Science*, pages 355–360. Springer, 2001.
2. 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.
3. D. J. Bagert and D. K. Friesen. A multi-language syntax-directed editor. In P. Davis and V. McClintock, editors, *Proceedings of the 15th ACM Annual Conference on Computer Science, St. Louis, Missouri, USA, February 16-19, 1987*, pages 300–302. ACM, 1987.
4. M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *GPCE*, pages 3–12, 2007.

5. 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.
6. M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax. In R. Glck and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallinn, Estonia, 2005. Springer.
7. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383, 2004.
8. M. Bravenboer and E. Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In H. Giese, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*, pages 34–46. Springer, 2007.
9. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
10. S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm, 2004.
11. K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.
12. S. Erdweg, L. C. L. Kats, Rendel, C. Kastner, K. Ostermann, and E. Visser. Growing a Language Environment with Editor Libraries. In *GPCE*, 2011.
13. S. Erdweg, T. Rendel, C. Kstner, and K. Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *OOPSLA*, 2011.
14. R. E. Faith, L. S. Nyland, and J. Prins. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In *DSL*, 1997.
15. M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005.
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
17. L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information & Software Technology*, 46(7):465–472, 2004.
18. G. Hedin and E. Magnusson. JastAdd–an aspect-oriented compiler construction system. *SCP*, 47(1):37–58, 2003.
19. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN*, 24(11):43–75, 1989.
20. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE*, pages 137–148, 2008.
21. P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse, ICSR '98*, jun 1998.
22. G. L. S. Jr. Growing a Language. *lisp*, 12(3):221–236, 1999.
23. L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. (best student paper award).
24. L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *OOPSLA*, pages 918–932, 2010.
25. P. Klint. A Meta-Environment for Generating Programming Environments. *TOSEM*, 2(2):176–201, 1993.
26. P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009.
27. H. Krahn, B. Rumpe, and S. Vlkel. MontiCore: a framework for compositional development of domain specific languages. *STTT*, 12(5):353–372, 2010.
28. R. T. Lindeman, L. C. L. Kats, and E. Visser. Declaratively Defining Domain-Specific Language Debuggers. In E. Denney and U. P. Schultz, editors, *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE 2011)*, pages 127–136, New York, NY, USA, 2011. ACM.

29. B. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
30. Y. Mali and E. V. Wyk. Building Extensible Specifications and Implementations of Promela with AbleP. In A. Groce and M. Musuvathi, editors, *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, volume 6823 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2011.
31. R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, 7(5):472–482, 1981.
32. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706. ACM, 2006.
33. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
34. M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. LISA: An Interactive Environment for Programming Language Development. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2002.
35. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.
36. D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2):91–105, 1985.
37. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *cc*, pages 138–152, 2003.
38. T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
39. S. W. Porter. Design of a syntax directed editor for psdl. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1988.
40. D. Rebernak, M. Mernik, H. Wu, and J. G. Gray. Domain-specific aspect languages for modularising crosscutting concerns in grammars. *IEE Proceedings - Software*, 3(3):184–200, 2009.
41. L. Renggli, T. Girba, and O. Nierstrasz. Embedding Languages Without Breaking Tools. In *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming,*, 2010.
42. T. W. Reps and T. Teitelbaum. The Synthesizer Generator. In *sde*, pages 42–48, 1984.
43. A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 199–210. ACM, 2009.
44. C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006.
45. C. A. Szyperski. *Component software - beyond object-oriented programming*. Addison-Wesley-Longman, 1998.
46. M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-Based Macro System for Java. In *oorase*, pages 117–133, 1999.
47. M. Voelter. Implementing feature variability for models and code with projectional language workbenches. 2010.
48. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Accepted for Publication at SPLASH/Wavefront 2012*, 2012.
49. M. Voelter and E. Visser. Product Line Engineering using Domain-Specific Languages. In *Software Product Line Conference*, 2011.
50. E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, 2008.

51. E. V. Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *CC*, pages 128–142, 2002.
52. E. V. Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute Grammar-Based Language Extensions for Java. In *ECOOP*, pages 575–599, 2007.