Language and IDE Modularization and Composition with MPS

Markus Voelter

Oetztaler Strasse 38, Stuttgart, Germany
voelter@acm.org

Abstract. Modularization and composition of languages and their IDEs is an important building block for working efficiently with domain-specific languages. Traditionally this has been a challenge because many grammar formalisms are not closed under composition, hence syntactic composition of languages can be challenging. Composing semantics can also be hard, at least in the general case. Finally, a lot of existing work does not consider IDEs for the composed languages. This paper illustrates how JetBrains MPS addresses language and IDE modularization and composition based on a projectional editor and modular type systems and transformations. The paper also classifies composition approaches according to the dependencies between the composed languages and whether syntactic composition is supported. Each of the approaches is illustrated with an extensive example implementation in MPS.

1 Introduction

Programmers typically use general purpose languages (GPLs) for developing software systems. The term *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.

One reason is the rising level of complexity of target platforms. 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 multi-language programming is the 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, user interface specification or scheduling) or a business domain (such as insurance contracts, refrigerator cooling algorithms or state-based programs in embedded systems). DSLs support these domains more efficiently than GPLs because they provide

linguistic abstractions for common idioms encountered in these domains. Using custom linguistic abstractions makes the code more concise, more suitable for formal analysis, verification, transformation and optimization, and more accessible to non-programmer domain experts.

The combined use of multiple languages in a single system raises the question of how the syntax, semantics, and the development environments (IDEs) of the various languages can be integrated. As we discuss in Section 6, each of these aspects has its own challenges and has been addressed to various degrees. Syntactic composition has traditionally been hard [26]. In particular, retaining decent IDE support (such as code completion, syntax coloring, static error checking, refactoring or debugging) in the face of syntactically composed languages is a challenge and hence is often not supported for a particular combination of languages. In some rare cases, syntactic integration between *specific* pairs of languages has been built, for example, embedded SQL in Java [31]. A more systematic approach for language and IDE modularization and composition is required. Language and IDE modularization and composition addresses 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 (constraints and type system) have to be integrated. For example, existing operators have to be overridden for new types.
- 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 or interpreters.
- Finally, the IDE that provides code completion, syntax coloring, static checks and other relevant services has to be composed as well.

In this paper we focus on JetBrains MPS[1] to demonstrate language composition approaches. MPS is a projectional editor and no grammars or parsers are used. Instead, editing gestures *directly* modify the abstract syntax tree (AST), and the representation on the screen is projected from the changing AST. Consequently, MPS' main contribution to language composition addresses the syntax and IDE aspects. However, MPS also provides good support for composing the static semantics and execution semantics of languages.

1.1 Contribution and Structure of the paper

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. We emphasize syntax and IDE, but also discuss type systems and transformation. While other, parser-based approaches can do language composition to some extent as

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

well, it is especially simple to do with projectional editors. So our third contribution is an implicit illustration of the benefits of using projectional editors in the context of language composition, based on the MPS example.

The paper is structured as follows. In Section 1.3 we define terms and concepts used throughout the paper. Section 1.4 introduces the four composition approaches discussed in this paper, and provides a rationale why we discuss those four approaches, and not others. We then explain how projectional editors work in general, and how MPS works specifically (Section 2). We develop the core language which acts as the basis for the modularization and composition examples in Section 3. This section also serves as a brief tutorial on language definition in MPS. The main part of the paper is Section 4 which shows the implementation of the four composition approaches in MPS. Section 5 discusses what works well and at what could be improved in MPS with regards to language and IDE modularization and composition. We conclude the paper with related work (Section 6) and a short summary.

1.2 Additional Resources

The example code used in this paper can be found at github[2] and works with MPS 2.0. A set of screencasts that walk through the example code is available on Youtube[3]. This paper is not a complete MPS tutorial. We refer to the Language Workbench Competition (LWC 11) MPS tutorial[4] for details.

1.3 Terminology

Programs are represented in two ways: concrete syntax (CS) and abstract syntax (AS). Users use the CS as they write or change programs. The AS is a data structure that contains all the data expressed with the CS, but without the notational details. The AS is used for analysis and downstream processing of programs. A language definition CS and AS, as well as rules for mapping one to the other. *Parser-based* systems map the CS to the AS. Users interact with a stream of characters, and a parser derives the abstract syntax tree (AST) by using a grammar. *Projectional* editors go the other way round. User editing gestures directly change the AST, the concrete syntax being a mere projection that looks (and mostly feels) like text. MPS is a projectional editor.

The AS of programs is a primarily a tree of program *elements*. Every element (except the root) is contained by exactly one parent element. Syntactic nesting of the CS corresponds to a parent-child relationship in the AS. 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. Each fragment f is a standalone AST. In file-based tools, a fragment corresponds to a file. E_f is the set of program elements in a fragment.

[2] https://github.com/markusvoelter/MPSLangComp-MPS2.0
[3] http://www.youtube.com/watch?v=lNMRMZk8KBE
[4] http://code.google.com/p/mps-lwc11/wiki/GettingStarted

A language l defines a set of language *concepts* C_l and their relationships. We use the term concept to refer to CS, AS plus the associated type system rules and constraints as well as a definition of its semantics. In a fragment, each program element e is an instance of a concept c defined in a 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: $lo(concept) \Rightarrow language$. 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 (short for *children*) is the set of parent-child relationships in a fragment f. Each $c \in Cdn$ has the properties *parent* and *child*. $Refs_f$ (short for *references*) 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 [30] 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 (short for *inheritances*) is the set of inheritance relationships for a language l.

An important concern in language and IDE modularization and composition is the notion of independence. An *independent language* does not depend on other languages. It can be defined as follows:

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

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

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

An *independent fragment* is one where all references stay within the fragment:

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

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

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

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

As elaborated by Harel and Rumpe in [19] the execution semantics of a language l are defined by mapping the syntactic constructs of l to concepts from the semantic domain S of the language. Different representations of S and the mapping $l \rightarrow S$ exist. Harel and Rumpe prefer to use mathematical formalisms as S because their semantics is well known, but acknowledge that other formalisms are useful as well. In this paper we consider the semantics of a language l to be defined via a *transformation* that maps a program expressed in l 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, and, in particular, we do not discuss interpreters as an alternative to transformations. This decision is driven partly by the fact that, in our experience, transformations are the most widely used approach for defining execution semantics.

The paper emphasizes *IDE* modularization and composition in addition to *language* modularization and composition. 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. Most of these are supported by MPS in a modular and composable way (the exceptions are profiling, which is not supported, and debugging, which is supported but on a too low-level of abstraction), we do not discuss those aspects in this paper to keep the paper at a reasonable length.

1.4 Classification of Composition Approaches

In this paper we 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 remainder of the paper.

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

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

■ *Reuse.* Reuse is similar to Referencing in that the respective programs reside in separate fragments and only references connect those fragments. 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.* Embedding combines the syntactic integration introduced by Extension with not having dependencies introduced by Reuse: *independent* languages can be used in the same *heterogeneous* fragment. An example is embedding a reusable expression language into another DSL. Since neither of the

two composed languages can have direct dependencies, the same expression language can be embedded into *different* DSLs, and a specific DSL could integrate *different* expression languages.

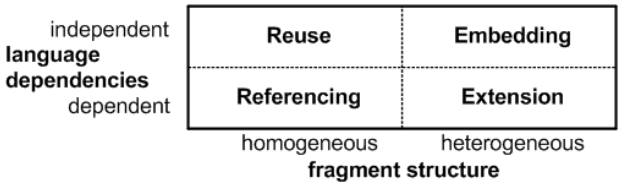

Fig. 1. We distinguish the four modularization and composition approaches regarding fragment structure and language dependencies. The dependencies dimension captures whether the languages have to be designed specifically for a specific composition partner or not. Fragment structure captures whether the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

As can be seen from the above descriptions, we distinguish the four approaches regarding fragment structure and language dependencies, as illustrated in Fig. 1 (other classifications have been proposed, they are discussed in Section 6). 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 them essential for the following reasons. *Language dependencies* capture whether a language has to be designed with knowledge about a particular composition partner in order to be composable with that partner. It is desirable in many scenarios that languages be composable *without* previous knowledge about possible composition partners. *Fragment Structure* captures whether the composed languages can be syntactically mixed. Since modular concrete syntax can be a challenge, this is not always easily possible, though often desirable.

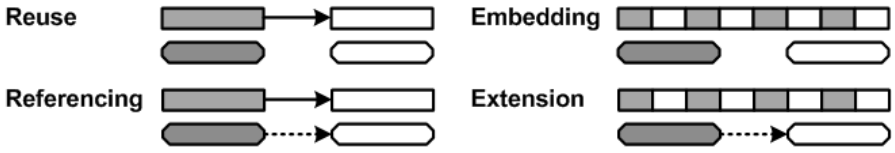

Fig. 2. The relationships between fragments and languages in the four composition approaches. Boxes represent fragments, rounded boxes are languages. Dotted lines are dependencies, solid lines references/associations. The shading of the boxes represent the two different languages.

1.5 Case Study

In this paper we illustrate the language and IDE modularization and composition approaches with MPS based on a set of example languages. At the center

is a simple `entities` language. We then build additional languages to illustrate the composition approaches introduced above (Fig. 3). 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. We also show Extension by extending MPS' built in BaseLanguage, a variant of Java.

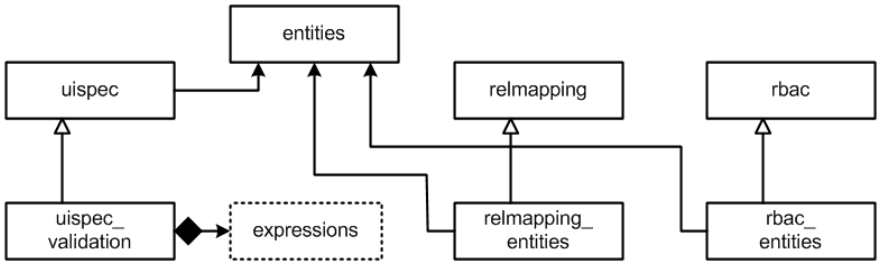

Fig. 3. `entities` is the central language. `uispec` defines UI forms for the `entities`. `uispec_validation` adds validation rules, and embeds a reusable expressions language. `relmapping` provides a reusable database mapping language, `relmapping_entities` adapts it to the `entities` language. `rbac`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

2 How MPS works

The JetBrains Meta Programming System[5] 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 [16]. 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.

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). In those tools 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

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

the model structure directly. While the CS 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. MPS achieves this quite well (it is beyond the scope of this paper to describe how). The following is a list of benefits of projectional editing:

- 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.
- Graphical, symbolic, tabular and textual notations can be mixed and combined, and they can be defined with the same formalism and approach. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions[6].
- Since projectionally defined languages always need an IDE for editing (to do the projection), language definition and composition always implies IDE definition and composition. The IDE provides code completion, error checking and syntax highlighting for any valid language, composed or not.
- Since 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). Examples of the latter 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 customizations to enable an editing experience that resembles modern IDEs that use automatically expanding code templates. This makes editing in MPS 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 AST. 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 CS. A final drawback is that MPS is not based on any industry standards. For example, it does not rely on EMF[7] or another widely used modeling formalism. However, since MPS' meta-metamodel is extremely close to EMF Ecore, it is trivial to

[6] Intentional's Domain Workbench has demonstrated this repeatedly, for example in [48]. As of 2012, MPS can do text, symbols (such as big sum signs or fraction bars) and tables. Graphics will be supported in 2013.

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

build an EMF exporter. Also, all other language workbenches also do not support portability of the *language definition* beyond the AS— which is trivial in terms of implementation effort.

MPS has been designed 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, composition of the CS 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 [26]. As we will see, this is not the case in MPS. Arbitrary languages can be combined syntactically.

■ *Extensible Type Systems.* All composition techniques require some degree of type system extension or composition[8]. MPS' type system specification is based on declarative typing rules that are executed by a solver. This way, additional typing rules for additional language concepts 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 must 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 configured to run *before* an existing transformation.

The examples 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 MPS. Like other language workbenches, MPS comes with a set of DSLs for language definition, a separate DSL for each language aspect such as structure, editor, type systems, generators as well as things like quick fixes or refactorings. MPS is bootstrapped, so these DSLs are built (and can be extended) with MPS itself.

We illustrate language definition with MPS based on a simple `entities` language. Example code is shown below. *Modules* are root nodes that live as top-level elements in *models*. According to the terminology introduced in Section 1.3 root nodes (and their descendants) are considered *fragments*.

[8] Note that the term *type system* really just refers to the actual type calculation and checks, as well as other constraints on the program. Resolution of symbol names is handled differently and not part of the type system.

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

```
// continued...
entity Department {
  id : int
  description : string
}
```

■ *Structure and Syntax.* Language definition starts with the AS, referred to as *structure* in MPS. Fig. 4 shows a UML diagram of the `entities` language AS. The following code shows the definition of the `Entity` concept[9]. `Entity` extends `BaseConcept`, the top-level concept similar to `java.lang.Object` in Java. It implements the `INamedConcept` interface to inherit a `name` property. 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
```

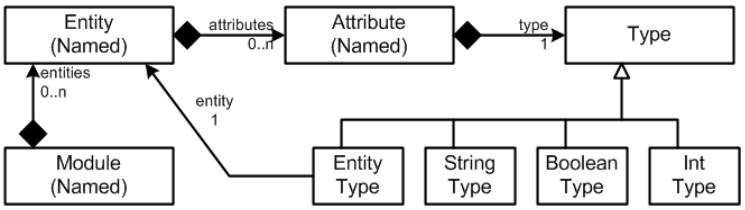

Fig. 4. The abstract syntax of the entities language. An `Entity` has `Attributes` which have a `Type` and a `name`. `EntityType` extends `Type` and references `Entity`. This adapts entities to types (cf. the Adapter pattern [18]). Concepts like `EntityType` which have exactly one reference are called smart references and are treated specially by MPS: instead of proposing to explicitly instantiate the reference concept and then selecting the target, the code completion menu shows *the possible targets* of the reference directly. The reference concept is implicitly instantiated once a target is selected.

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. Fig. 5 explains the editor for `Entity`. The editors for the other concepts are defined similarly.

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

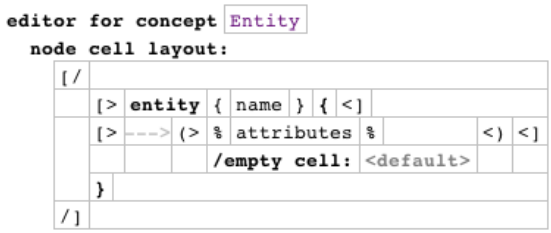

Fig. 5. The editor for `Entity`. The outermost cell is a vertical list `[/ .. /]`. In the first line, we use a horizontal list `[> .. <]` that contains the keyword `entity`, the `name` property and an opening curly brace. In the second line we use indentation `-->` and a vertical arrangements of the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

■ *Type System.* As we have explained in Section 2, language developers specify typing rules for language concepts. To calculate and check types for a program, MPS "instantiates" these rules for each instance of the concept, resulting in a set of type equations for a program. These equations contain type values (such as `int`) as well as type variables (for program elements whose type has not yet been calculated). MPS' then solves the set of type equations, trying to assign type values to the type variables in a way such that all the equations for a program are free of contradictions. If a contradiction arises, this is flagged as a typing error. For the `entities` language, we specify two simple typing rules. The first one specifies the type of a `Type` (it answers the questions of what the type system's type of should be for e.g. `int` if it is used in an attribute such as `int age;`):

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

This rule has the name `typeof_Type` and applies to the language concept `Type` (`int` or `string` are subconcepts of the abstract `Type` concept). The `typeof(...)` operator creates a type variable associated with a program element, in this case, with an instance `t` of `Type`. We calculate that type by cloning `t` itself. In other words, if the type system engine needs to find out the type of an `int` program element, that type is `int` as well. This may be a bit confusing, because instances of `Type` (and its subconcepts) play two roles. First, they are part of the program itself if they are explicitly specified in attributes (`int age;`). Second, they are also the objects with which the type system engine works. By cloning the program element, we express that types represent themselves in the type system engine (we need a clone since for technical reasons we cannot return the program element itself). Another typing rule defines the type of the `Attribute` as a whole to be the type of the attribute's `type` property:

```
rule typeof_Attribute for Attribute as a {
    typeof(a) :==: typeof(a.type); }
```

This rule answers the question of what the type system engine's type should be for instances of `Attribute`. Note how this equation has two type variables and no type values. It simply propagates the type of the attribute's specified type (the `int` in `int age;`) to be the type of the overall attribute. Note how the two typing rules discussed in this section play together to calculate the type of the `Attribute` from the type specified by the user: First we calculate the type of the specified type (a clone of itself) and then we propagate this type to the type of the whole attribute.

There is one more rule we have to define. It is not strictly part of the type calculation. It is a constraint that checks the uniqueness of attribute names for any given `Entity`:

```
checking rule check_Entity for Entity as e {
     set<string> names = new hashset<string>;
     foreach a in e.attributes {
         if (names.contains(a.name)) {
             error "duplicate attribute name" -> a;
         }
         names.add(a.name);
     }
}
```

This rule does *not* establish typing equations, it just checks a property of the program (note the *checking* in the rule header). It checks attribute name uniqueness based on a set of the names. It reports an error if it finds a duplicate. It annotates the error with the attribute `a`, so the editor can highlight the respective program element. Note how in case of the typing rules shown above we don't have to perform the check and report an error ourselves. This is done implicitly by the type system engine.

■ *Generator.* From `entities` models we generate Java Beans. Since Java is available in MPS (called 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 maps ASTs onto other ASTs. Alternatively, one can use an 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.

MPS templates look like text generation templates known from tools such as Xpand[10], Jet[11] or StringTemplate[12] since they use the CS of the target language in the template. However, that CS is projected like any other program, and the IDE can provide support for the target language *in the template* (we discuss details on support for the target language in templates in Related Work, Section 6). This also means that the *template code itself* must be valid in terms of the target language.

[10] http://www.eclipse.org/modeling/m2t/?project=xpand
[11] http://www.eclipse.org/modeling/m2t/?project=jet
[12] http://www.stringtemplate.org/

Template-based generators consist of mapping configurations and templates. Mapping configurations define which elements are processed by which templates. For the `entities` language, we need a root mapping rule and reduction rules. *Root mapping rules* create new root nodes from existing root nodes (they a fragment to another fragment). 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, it replaces the element with the result of the associated template. In our case we reduce the various types (`int`, `string`, etc.) to their Java counterparts. Fig. 6 shows a part of the `entities` mapping configuration.

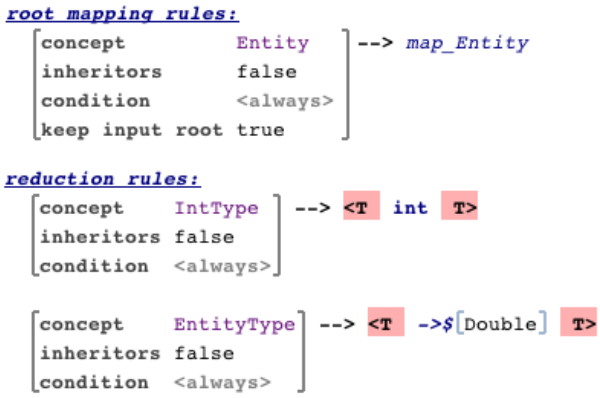

Fig. 6. The mapping configuration for the `entities` language. The root mapping rule for `Entity` specifies that instances of `Entity` should be transformed with the `map_Entity` template (which produces a Java class and is shown in Fig. 7). The reduction rules use inline templates, i.e. the template 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.

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`:

- Developers first write 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 field to a class (see `aField` in Fig. 7).
- Then macros are attached to those program elements in the example code that have to be replaced with elements from the input model during the transformation. 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). This expression returns the set of `Attribute`s from the current `Entity`, making the `LOOP` iterate over all attributes of the entity and create a field for each of them.

– At this point, each created field would be *identical* to the example code to which we attached the `LOOP` macro (`private int aField;`). 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 to the name of the `Attribute` we currently transform.

```
[root template
input Entity]
public class $[map_Entity] extends <none> implements <none>
  $LOOP$[private $COPY_SRC$[int] $[aField]; ]
  public map_Entity() {
  }
  $LOOP$[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 `Attribute`s of the `Entity`. The running text explains the details.

Instead of mixing template code and target language code (and separating them with some kind of escape character) we annotate macros to regular, valid target language code. Macros can be attached to arbitrary program elements. 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 generic MPS mechanism not specific to transformation macros and are discussed in Section 4.5.

4 Language Composition with MPS

In this section we discuss the four language and IDE modularization and composition techniques introduced in Section 1.4, plus an additional one that works only with a projectional editor such as MPS. For the first four, we provide a concise prose definition plus a set of formulas. We then illustrate each technique with a detailed MPS-based example based on the `entities` language.

4.1 Language Referencing

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

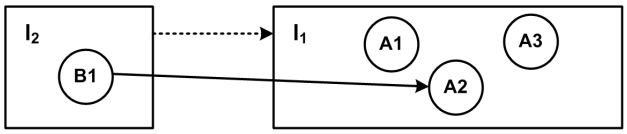

Fig. 8. Referencing: Language l_2 depends on l_1, because concepts in l_2 reference concepts in l_1. (We use rectangles for languages, circles for language concepts, and UML syntax for the lines: dotted = dependency, arrows = associations, hollow-triangle-arrow for inheritance.)

A fragment f_2 depends on f_1. f_2 and f_1 are expressed with languages l_2 and l_1, respectively. We call l_2 the *referencing* language, and l_1 the *referenced* language. 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. While equations (2) and (3) (from Section 1.3) continue to hold, (1) does not. Instead

$$\forall r \in \mathit{Refs}_{l_2} \mid lo(r.\mathit{from}) = l_2 \ \wedge \ (lo(r.to) = l_1 \vee lo(r.to) = l_2) \qquad (7)$$

From a CS perspective, such a reference is a simple identifier, (possibly with dots). This terminal can easily be redefined in the referencing language and does not require reusing and embedding non-terminals from the referenced language. Hence no syntactic composition is required in this case.

As an example, for Referencing we define a language `uispec` for defining UI forms for `entities`. Below is an example. This is a *homogeneous* fragment, expressed only in the `uispec` language. Only the identifiers of the referenced elements (such as `Employee.name`) have been added to the referencing language as discussed in the previous paragraph. However, the fragment is *dependent*, since it references elements from another fragment (expressed in the `entities` 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
```

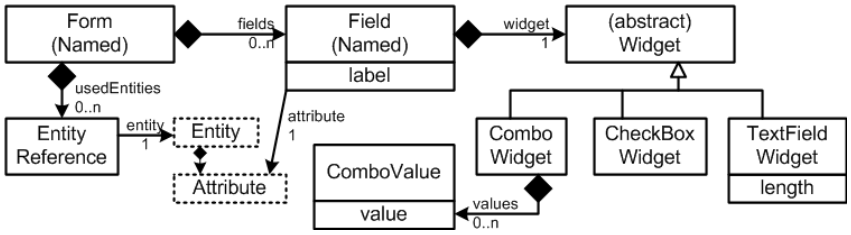

Fig. 9. The abstract syntax of the `uispec` language. Dotted boxes represent classes from another language (here: the `entities` language). A `Form` contains `EntityReference`s that connect to an `entities` model. A `Form` also contains `Field`s, each referencing an `Attribute` from an `Entity` and containing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

■ *Structure and Syntax.* The AS for the `uispec` language is shown in Fig. 9. The `uispec` language extends[13] the `entities` language. This means that concepts from the `entities` language can be used in the definition of the `uispec` language. A `Form` owns a number of `EntityReferences`, which in turn reference an `Entity`. Below is the definition of the `Field` concept. It has a `label` property, owns a `Widget` and refers to the `Attribute` it edits.

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

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 ambiguities or 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. The typing rule below implements these checks and is defined in the `uispec` language. It references types from the `entities` language. We use a `checking rule` to illustrate how constraints can be written that do not use the inference engine introduced earlier.

[13] 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 {
  node<Widget> w = field.widget;
  node<Type> t = field.attribute.type;
  if (w.isInstanceOf(CheckBoxWidget) && !(t.isInstanceOf(BooleanType))) {
    error "checkbox can only be used with booleans" -> w;
  }
  if (w.isInstanceOf(ComboWidget) && !(t.isInstanceOf(StringType))) {
    error "combobox can only be used with strings" -> w;
} }
```

■ *Generator.* The defining characteristic of 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 example, the generated code exhibits the same separation. From a `Form` we generate a Java class that uses Java Swing to render the UI. It *uses* the Beans generated from the `entities`: they are instantiated, and the setters are called. The generators are separate but they are *dependent*, since the `uispec` generator knows about the names of the generated Java Beans, as well as the names of the setters and getters. This dependency is realized by defining a set of behaviour methods on the `Attribute` concept that are called from both generators (the colon in the code below represents the node cast operator and binds tightly; the code 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.toFirstUpper(); }
  public string getterName() { "get" + this.name.toFirstUpper(); }
}
```

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 of l_1, some of them typically extend concepts in l_1. While l_1 remains independent, l_2 is dependent on l_1:

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

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 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 Extension, because the approach would not be modular, it is invasive. Also, native interfaces that supports calling one language from another (such as calling C from Perl or Java) is not Extension; rather it is a form of language Referencing. The fragments remain homogeneous.

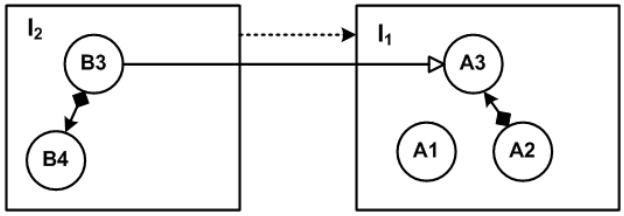

Fig. 10. Extension: l_2 extends l_1. It provides additional concepts $B3$ and $B4$. $B3$ extends $A3$, so it can be used as a child of $A2$, plugging l_2 into the context provided by l_1. Consequently, l_2 depends on l_2.

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 and color.

```
$LOOP$[->$[o].->$[split]( $COPY_SRC$[String] newValue = $SWITCH$ [null]; );
                          yield 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 expressions are used mostly when implementing generators; this screenshot shows a generator that uses a block expression.

A block expression is a block that can be used where an `Expression` is expected [6]. It can contain any number of statements; `yield` can be used to "return values" from the block. A block expression can be seen as an "inlined method" or a closure that is defined and called directly. The generator of the block expression from Fig. 11 transforms it into a method and a call to it:

```
aEmployee.setName( retrieve_name(aEmployee, widget0) );
...

public String retrieve_name(Employee aEmployee, JComponent w) {
  String newValue = ((JTextField) w).getText();
  return newValue;
}
```

■ *Structure and Syntax.* The `jetbrains.mps.baselanguage.exprblocks` language extends MPS' BaseLanguage. The block expression is used in places where the base language expects an `Expression`, so a `BlockExpression` extends `Expression`. Consequently, fragments that use the `exprblocks` language, can now use `BlockExpression`s in addition to the concepts provided by the base language. The fragments become *heterogeneous*.

```
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(aYield) :==: typeof(aYield.result)` (the type of `yield 1;` is `int`, because the type of `1` is `int`). Since the `BlockExpression` is used as an expression, it has to have a type as well: the type of the `BlockExpression` is the common super type of the types of all the `yield`s:

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

This equation iterates over all `yield` statements in a block expression and establishes an equation between the current `yield`'s type and a type variable `resultType`. It uses the `:>=:` operator to express that the `resultType` must be the same or a supertype of the type of each `yield`. The only way to make *all* of these equations true (which is what the type system solver attempts to do) is to assign the common supertype of all `yield` types to `resultType`. We then associate this `resultType` with the type of the overall block expression.

■ *Generator.* The generator reduces `BlockExpression`s to BaseLanguage. It transforms a heterogeneous fragment (BaseLanguage and `exprblocks`) to a homogeneous fragment (BaseLanguage only). The first step is the creation of the additional method for the block expression as shown in Fig. 12 and Fig. 13.

The template shown in Fig. 13 shows the creation of the method. The mapping label (`b2M`) 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).

Another concept introduced by the `exprblocks` language is the `PlaceholderStatement`. It extends `Statement` so it can be used in function bodies. It is used to mark locations at which subsequent generators can 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.

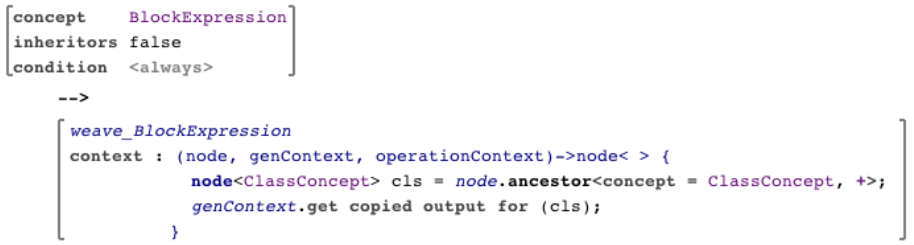

Fig. 12. We use a weaving rule to create an additional method for a block expression. A weaving rule processes an input element (a `BlockExpression`) by creating another element *in a different location*. The `context` function defines the target location. In this example, it simply gets the class in which we have defined the particular block expression, so the additional method is generated into that same class. The called template `weaveBlockExpression` is shown in Fig. 13.

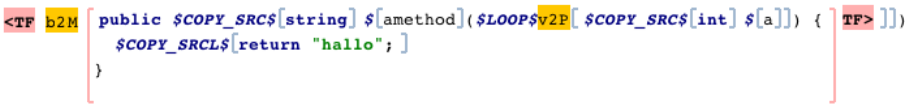

Fig. 13. This generator template creates a method from the block expression. It uses `COPY_SRC` macros to replace the dummy `string` type with the computed return type of the block expression, inserts a computed name, adds a parameter for each referenced variable outside the block, and inserts all the 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).

In the classification (Section 1.4) we mentioned that we consider language restriction as a form of Extension. To illustrate this point 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 `BlockExpression`. The constraint expression has to return `true` if the respective `childConcept` is valid in this location. We return true if 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 of the child. This is important to keep dependencies pointing in the right direction.

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

Fig. 14. Here we generate the call to the method generated in Fig. 13. We use the mapping label `b2M` to refer to the correct method (not shown; happens inside the reference macro). We pass in the variables from the call's environment as actual arguments using the `LOOP` and `COPY_SRC` macros.

Extension comes in two flavors. One feels like Extension, and the other one feels more like Embedding. In this section we have described the Extension flavor: 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 use some of the syntax provided by a base language in that new language. For example, we could create a state machine language that reuses Java's expressions in guard conditions. This use case *feels* like Embedding (we embed syntax from the base language in our new language), but in terms of our classification (Section 1.4) it is still Extension. Embedding would prevent a dependency between the state machine language and Java.

4.3 Language Reuse

Language Reuse 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

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

Since l_2 is independent, its concepts 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.

A way of realizing dependent fragments with independent languages is using an adapter language l_A (cf. [18]) that contains concepts that *extend* concepts in l_2 and *references* concepts in l_1 (Fig. 15). One could argue that in this case Reuse is just a combination of Referencing and Extension. This is true from an implementation perspective, but it is worth describing as a separate approach because it enables the combination of two *independent languages* with an adapter *after the fact*, so no pre-planning during the design of l_1 and l_2 is necessary.
Reuse covers the case where a language has been developed independent of its reuse context. The respective fragments remain homogeneous. We cover two alternative cases: in the first one (a persistence mapping language) the generated code is separate from the code generated from the `entities` language. The second one (a language for role-based access control) describes the case where the generated code has to be "woven into" the `entities` code.

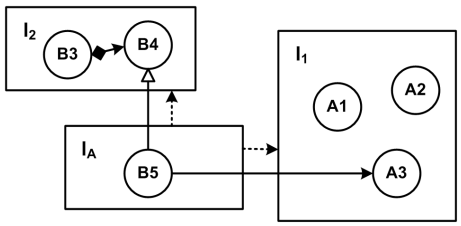

Fig. 15. Reuse: l_1 and l_2 are independent languages. Within an l_2 fragment, we still want to be able to reference concepts in a fragment expressed with l_1. To do this, an adapter language l_A is added that uses Extension and Referencing to adapt l_1 to l_2.

Separated Generated Code `relmapping` is a reusable language for mapping arbitrary data to relational tables. It supports the definition of relational table structures, but leaves the actual mapping to the source data unspecified. When the language is adapted to a specific context, this one mapping has to be provided. The left side if the code below shows the reusable part. A database is defined that contains tables with columns. Columns have (database-specific) data types. On the right side we show the database definition code when it is reused with the `entities` language; each column is mapped to an entity attribute.

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

```
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
```

■ *Structure and Syntax.* 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 in a context-specific way.

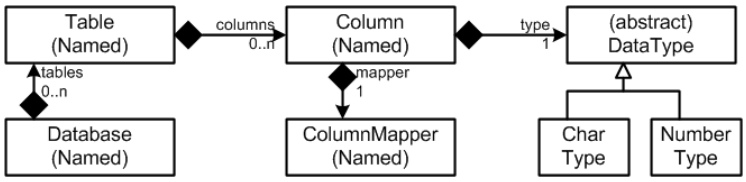

Fig. 16. A `Database` contains `Tables` which contain `Columns`. A column has a name and a type. A column also has a `ColumnMapper`. This is an abstract concept that determines where the column gets its data from. It is a hook intended to be specialized in sublanguages, specific to the particular Reuse 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 `relmapping` language projects the column mapper — and its context-specific sub-concepts — on the right of the field definition, resulting in heterogeneous fragments.

■ *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`. 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 implements this method by mapping `IntType` to `NumberType`, and everything else to `CharType`:

```
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 base class generated by the generator of the `relmapping` language and a context-specific subclass, generated by `relmapping_entities`. The generic base class contains code for creating the tables and for storing data in those tables. It contains abstract methods for accessing 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).

```
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) {
    Insert i = new Insert("Departments");
    i.add( "id", getValueForDepartments_id(applicationData));
    i.add( "descr", getValueForDepartments_descr(applicationData));
    i.execute();
```

```
  }

  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 (shown below) in the `relmapping_entities` language implements the abstract methods defined by the generic superclass. The interface, represented by the `applicationData` object, has to be generic so any kind of user data can be passed in. Note how this class references the Beans generated from the `entities`.

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

Interwoven generated code `rbac` is a language for specifying role-based access control for `entities`. The code below shows an example fragment. Note how it references entities (`Department`) and attributes (`Employee.name`) from an `entities` fragment.

```
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
```

■ *Structure and Syntax.* The structure is shown in Fig. 17. Like `relmapping`, `rbac` provides a hook `Resource` to adapt it to context languages. The sub-language `rbac_entities` provides two sub-concepts of `Resource`, namely `AttributeResource` to reference to an `Attribute`, and `EntityResource` to refer to an `Entity`, to define permissions for entities and their attributes.

■ *Type System.* No type system rules apply here, because none of the concepts added by the `rbac` language are typed or require constraints regarding the types in the `entities` language.

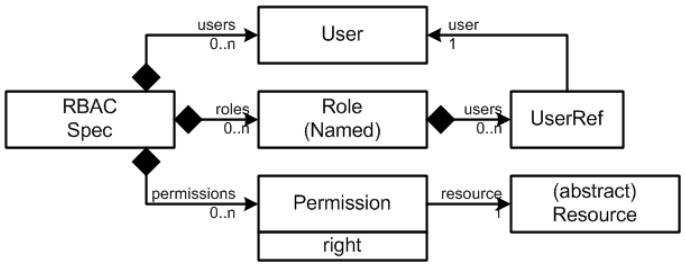

Fig. 17. Language structure of the `rbac` language. An `RBACSpec` contains `Users`, `Roles` and `Permissions`. Users can be members in several roles. A permission assigns a role and right (read, write) to a `Resource` (such as an `Entity` or an `Attribute`).

■ *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` (we cannot use the convenient base class/-subclass approach). Instead, a permission check is required *inside* the setters of the Java Beans. Here is some example code:

```
public void setName(String newValue) {
  if (new RbacSpecEntities().hasWritePermission("Employee.name")) {  // check permission
    throw new RuntimeException("no permission");                     // from rbac_entities
  }                                                                  // language
  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[14]. 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 do not use this approach here.
- An interceptor framework (see the Interceptor pattern in [9]) 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 do not use AspectJ in this paper.
- 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.

[14] http://www.eclipse.org/aspectj/

– 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 knowledge about the way the hook works.

Notice that only the AspectJ solution would work *without any pre-planning* from the perspective of the `entities` language, because it avoids mixing the generated code artifacts (it is handled by AspectJ). All other solutions require the original `entities` generator to "expect" extensions. In our case we have modified the `entities` generator to generate a `PlaceholderStatement` (Fig. 18) into the setters. 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 location, we do not have to predefine *how*.

```
$LOOP$[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`. If the generator encounters a placeholder (that has been put there by the `entities` generator) it replaces it with 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 before it can be replaced) and `strictly 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 a generator properties dialog. Note that specifying the priorities does not introduce additional language dependencies, modularity is retained.

4.4 Language Embedding

Language Embedding enables *heterogeneous* fragments with *independent* languages. It is similar to Reuse in that there are two independent languages l_1 and l_2, but instead of establishing references between two homogeneous fragments, we now *embed* instances of concepts from l_2 in a fragment f expressed with l_1:

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

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

Fig. 19. This reduction rule replaces `PlaceholderStatement`s 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.

Unlike language Extension, where l_2 depends on l_1 because concepts in l_2 extend 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 can be used to achieve this (we describe an Embedding *without* adapters in Section 4.5). However, in this case concepts in l_A don't just reference concepts from l_1. Instead, they contain them (similar to Fig. 15, but with a containment link between $B5$ and $A3$).

As an example we embed an existing `expressions` language into the `uispec` language without modifying either the `uispec` language or the expression language, since, in case of Embedding, none of them may have a dependency on the other. Below is an example program using the resulting language that uses expressions after 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 language `uispec_validation` that extends `uispec` and 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`.

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

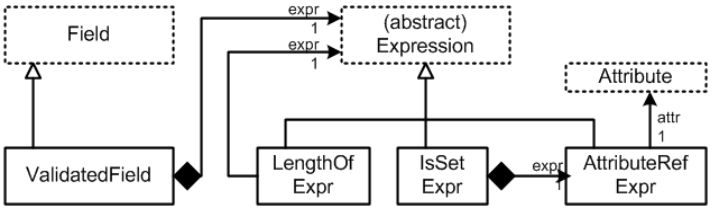

Fig. 20. The `uispec_validation` language defines a subtype of `uispec.Field` that contains an `Expression` from a reusable `expressions` language. The language also defines a couple of additional expressions, including the `AttributeRefExpr`, which can be used to refer to attributes of entities.

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 `Add Validation`[15]. This transforms an existing `Field` into a `ValidatedField`, so that a validation expression can be entered. The core of the Intention is the following script, which performs the actual transformation:

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

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 (the `uispec` and `entities` languages in the example). The following is the structure definition of the `LengthOfExpr`.

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

[15] We could alternatively also implement a way for people to just type `validate` on the right side of a field to trigger this transformation

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 similar to the leading keyword of the concept's CS. 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 attributes of entities that are used in the `Form` within which we define the validation expression. The code below defines the necessary scoping rule:

```
(model, scope, referenceNode, enclosingNode) -> 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);
  }
  return 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 add a child of type `Expression` to the `ValidatedField` concept.

■ *Type System.* 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 concepts, so the two sets of types must be mapped. For example, the type of the `IsSetExpression` is `expressions.BooleanType` so it fits in with the `expressions` language. The type of the `LengthOfExpr`, 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(attrRef) :==: typeof( attrRef.attr.type)`. However, consider the following code:

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

This code states that if the `worksAt` attribute of an employee is set, then its `freelancer` attribute must be `false` else it must be `true`. It uses the `==` 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 `==` operator. In the expressions language, we define overloaded operation rules. We specify the resulting type for an `Equals-`

`Expression` 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 `expressions.BooleanType`:

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

This overloaded operation specification is integrated with the inference-based typing rules using the following code:

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

To override these typing rules for `entities.BooleanType`, we simply provide another overloaded operation specification in the `uispec_validation` language:

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

■ *Generator.* For the generator we can use the following two alternative approaches. We can use the `expressions` language's existing to-text generator and wrap the expressions in some kind of `TextWrapperStatement`. A wrapper is necessary because we cannot simply embed text in BaseLanguage — this would not work structurally. Alternatively, we can write a (reusable) transformation from `expressions` to BaseLanguage; these rules would be used as part of the transformation of `uispec_validation` code to BaseLanguage. Since many DSLs will map code to BaseLangauge, it is worth the effort to write a reusable generator from `expressions` to BaseLanguage expressions. We choose this second alternative.

The actual expressions defined in the `expressions` language and those of BaseLanguage are almost identical, so this generator is trivial. We create a new language project `expressions.blgen` and add reduction rules. Fig. 21 shows some of these reduction rules.

We also need reduction rules for the new expressions added in the `uispec_validation` language (`AttrRefExpression`, `isSetExpression`, `LengthOfExpr`).

```
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 `expressions` 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.

Those rules 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.

```
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 reference macros (->$) to "rewire" the reference to the Java Bean instance, and the `toString` method call to a call to the getter.

Just as in the discussion on Extension (Section 4.2), we may want to use constraints to restrict the embedded language in the context of a `ValidatedField`. Consider the case where we wanted to embed the expressions part of C instead of the `expressions` language. C comes with all kinds of operators relating to pointers, bit shifting and other C-specifics that are not relevant in the validation 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 (the same discussion applied to the case where a base language is extended with *several independently developed extensions* at the same time). Let us consider the potential cases:

Same Concept Name: Embedded languages may define concepts with the same name as the host language. This will not lead to ambiguity because concepts have a unique 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 (users can always see the qualified name of the instantiated concept in the Inspector as a means of disambiguation).

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

In a projectional editor, the CS of a program is projected from the AST. A projectional system always goes from AS to CS, never from CS to AS (as parsers do). This has the important consequence that the CS does not have to contain all the data necessary to build the AST (which in case of parsers, is necessary). This has two consequences:

- A projection may be *partial*. The AS may contain data that is not shown in the CS. The information may, for example, only be changeable via intentions (see Section 4.4), or the projection rule may project some parts of the program only in some cases, controlled by some kind of configuration.
- It is also possible to project *additional* CS that is not part of the CS definition of the original language. Since the CS 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. It represents a variant of Embedding: no dependencies, but syntactic composition. The mechanism MPS uses for this is called annotations, which we have seen when we introduced templates (Section 3): an annotation can be attached to arbitrary program elements and can be shown together with CS 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 `entities`, as well as the annotations (e.g. `-> People.id`). From a CS perspective, the column mapping is embedded in the `Entity`. In the AST the mapping information is also actually stored in the `entities` model. However, the definition of the `entities` language does not know that this additional information is stored and projected "inside" entities. The `entities` language is not modified.

■ *Structure and Syntax.* 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
```

`AttrToColMapping` concept extends `NodeAnnotation`, a concept predefined by MPS. Concepts that extend `NodeAnnotation` have to provide a `role` property and an `annotated` concept link. As we have said above, 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 prepended with `@`. The `annotated` concept link points to the concept *to which this annotation can be added*. `AttrToColMapping`s can be annotated to instances of `Attribute`.

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

It is possible to define the annotation target to be `BaseConcept`, which means the annotation can be attached to *any* program element. This is useful for generic metadata such as documentation, requirements traces or presence conditions in

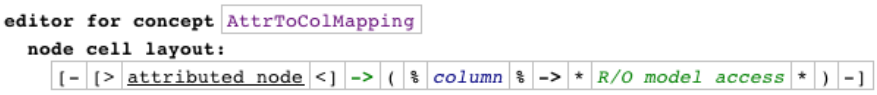

Fig. 23. The editor for the `AttrToColMapping` embeds the editor of the concept it is annotated to (using the `attributed node` cell). It then projects the reference to the referenced column. This way the editor of the annotation has control of if and how the editor of the annotated element is projected.

product line engineering (we describe this in [54] and [52]). MPS' template language uses this approach as well. Note that this is a way to support Embedding generically, *without* the use of an adapter language. The reason why this generic approach is useful mostly for metadata is related to semantics: since the annotations can be composed *with any other language* without an adapter, the semantics must be generic as well, i.e. not related to any particular target language. This is true for the generic metadata mentioned above.

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

■ *Generator.* The generator is also similar to one for `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

In this section we discuss limitations of MPS in the context of language and IDE modularization and compositionand discuss an approach for improving some of these shortcomings. We also look at real world use of MPS.

5.1 Limitations

The paper paints 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 missing features. and problems have been solved ad hoc as the they arose. A consistent, unified approach is sometimes missing. I propose such an approach in Section 5.2.

■ *Syntax.* The examples in this paper show that meaningful language and IDE modularization and composition is possible with MPS. 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 (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 CS could be solved by changing the notation (or color or font) of existing concepts if they are used together with a particular other language. Such a new CS would be defined in the respective adapter 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 language developer. However, there are additional concerns an IDE may address including version control integration, 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, although a profiler for language *implementations* is on the roadmap. MPS comes with a debugging framework that lets language developers create debuggers for languages defined in MPS. 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 [53] that develops an extensible version of the C programming language on top of MPS we have developed a framework for extensible C debuggers. Developers of C extensions can easily specify how the extension integrates into the C debugger so that debugging on the *syntax of the extension* becomes possible for heterogeneous fragments. We are currently in discussions with JetBrains to make the underlying extensible debugging framework part of MPS. Debuggers for DSLs also been discussed by Visser et al. in [29] and by Wu et. al. in [55].

■ *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/extending/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 they depend on.

■ *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.* Language designers specify a partial ordering among generators using priorities. It is not easily possible to "override" an existing generator, but generators can run *before* or *after* 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 pre-planning on the part of the developer of the generator that adds the placeholder.

5.2 A unified approach

Looking at the limitations discussed in the previous subsection it is clear that a consistent approach for addressing the modularization, extension and composition of *all* language aspects would be useful. In this section we propose such a unified approach based on the principles of component-based design [50]. In this approach, all language aspects would use components as the core structural building block. Components have facets and a type. 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. 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. To support modularization, 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 an element before the original generator gets a chance to reduce it.
- *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 exchangeable.
- *overriding* means that the original facet is completely shadowed by the new one. This could be used to define a new editor for an existing concept.

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 pre-planning 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 sub-components. A language is abstract as long as it has at least one abstract component, for which no concrete sub-component is provided. Component parameters could even be usable in structure definitions, for example as the base concept; this would make a language extension parameterizable regarding the base language it extends.

5.3 Real-World use of MPS

The examples in this paper are toy examples — the simplest possible languages that can illustrate the composition approaches. However, MPS scales to realistic

systems, both in terms of language complexity and in terms of program size. The composition techniques — especially those involving 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 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 [20]), 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[16] addresses these challenges using incremental, modular extension of C with domain-specific language concepts. mbeddr uses Extension to add interfaces and components, state machines, and 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 than 10.000 lines of MPS code. Scalability tests have shown that mbeddr 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 [53].

■ *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 Embedding is used to provide an integrated web development environment[17].

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. *dnq* also includes a collections language which supports the manipulation of collections in a way similar to .NET's Linq [34]. Other languages include *webr*, a language used for implementing interactions between 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*. 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.

[16] http://mbeddr.com
[17] http://www.jetbrains.com/mps/docs/MPS_YouTrack_case_study.pdf

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

6 Related Work

This paper addresses language and IDE modularization and composition with MPS, a topic that touches on many different topics. In this section we discuss related work focusing on modular grammars and parsers, projectional editing, modular compilers and modular IDEs. We conclude with a section on related work that does not fit these categories.

6.1 Modular Grammars and Parsers

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

In [26] Kats, Visser and Wachsmut describe nicely the trade-offs with non-declarative grammar specifications and the resulting problems for composition of independently developed grammars. Grammar formalisms that cover only subsets of the class of context-free grammars are not closed under composition: resulting grammars are likely to be outside 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. In [47] Schwerdtfeger and van Wyk also discuss the challenges in grammar composition. They also describe a way of verifying early (i.e. before the actual composition attempt) whether two grammars are composable or not.

An example of a grammar formalism that supports the full set of context-free grammars is the Syntax Definition Formalism [22]. SDF is implemented with scannerless GLR parsers. Since it parses tokens and characters in a context-aware fashion, there will be 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 [31]). 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 grammar that defines the composition of two other independent grammars (discussed in [7]). The SILVER/COPPER system described by van Wyk in [56] solves these ambiguities via disambiguation functions written specifically for each combination of ambiguously composed grammars. Note that in MPS such disambiguation is never necessary. We discuss the potential for ambiguity and the way MPS 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 in AbleJ [58] and and for SPIN's Promela language in AbleP [32]. A similar approach is discussed for an SDF-based system in [8]. However, ad-hoc inclusion only works as long as the set of included extensions (which have presumably been developed independent from each other) are not ambiguous with regards to each other. In case of ambiguities, disambiguations have to be defined as described above.

Polyglot, an extensible compiler framework for Java [40] also uses an extensible grammar 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 remains in the LALR subclass.

In Section 3 we mentioned that MPS' template language provides IDE support for the target language *in the template*. In traditional text-generation template languages this is typically not supported because it requires support for language composition: the target language must be embedded in the template language. However, there are examples of template languages that support this. Not surprisingly they are built on top of modular grammar formalisms. An example is the Repleo template language [1] which is built on SDF. However, as explained in the discussion on SDF above, SDF requires the definition of an additional grammar that defines how the host grammar (template language in this case) and the embedded grammar (target language) fit together: for all target language non-terminals where template code should be allowed, a quotation has to be defined. MPS does not require this. Any target language can be marked up with template annotations. No separate language has to be defined for the combination of template language and target language.

6.2 Projectional Editing

Projectional editing (also known as structural editing) is an alternative approach for handling the relationship between CS and AS, i.e. it is an alternative to parsing. As we have seen, it simplifies modularization and composition.

Projectional editing is not a new idea. An early example is the Incremental Programming Environment (IPE, [33]). It uses a structural editor for users to interact with the program and then incrementally compiles and executes the resulting AST. 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 example, 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 be aware of the language structure at all times. MPS in contrast goes a long way in supporting editing gestures that much more resemble text editing, particularly for expressions. IPE also does not address language modularity. In fact it comes with a fixed, C-like language and does not have a built-in facility to define new languages. It is not bootstrapped. Another projectional system is GANDALF [39]. Its ALOEGEN component generates projectional editors from

a language specification. It has the same usability problems as IPE. This is nicely expressed in [42]: *Program editing 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 [45] also supports structural 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 *at the expression level* is prohibited. MPS does not use this "trick", and instead supports projectional editing also on expression level, with convenient editing gestures. We have seen in this paper that extensions of expressions are particularly important to tightly integrate an embedded language with its host language.

Bagert and Friesen describe a multi-language syntax directed editor in [4]. However, this tool supports only Referencing, syntactic composition is not supported.

The Intentional Domain Workbench [48] is another contemporary projectional editor that has been used in real projects. An impressive demonstration about its capabilities can be found in an InfoQ presentation titled Domain Expert DSL[18].

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.

Many 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). Example of systems that make use of attribute grammars for type computation and type checking include SILVER ([56], mentioned above), JastAdd [21] and LISA ([36], discussed in more detail in the next section). Forwarding (introduced in [57]) is a mechanism that improves the modularity of attribute grammars by delegating the look-up of an attribute value to another element.

While MPS' type system specification language can be seen as associating a type attribute with AST elements using the `typeof` function, MPS' type system is different from attribute grammars. Attribute values are calculated by *explicitly* referring to the values of other attributes, often recursively. MPS' type system rules are declarative: users specify typing rules for language concepts and MPS "instantiates" each rule for each AST element. 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.

As we have seen, for language Extension the execution semantics is defined via transformation to the base language. In [56], van Wyk discusses under which circumstances such transformations are valid: the changes to the overall AST must be local. No global changes are allowed to avoid unintended interactions

[18] http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk

between several independently developed extensions used in the same program. In MPS such purely local changes are performed with reduction rules. In our experience, it is also feasible to *add* additional elements to the AST *in select places*. 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 of defining 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 [38] 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, in [43] Rebernak el al. 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 do not support aspect-oriented extension of the concrete syntax.

6.4 Modular IDEs

Based on the fundamentals that enable modular syntax and semantics we now look at tools that, from a language definition, also create a language-aware editor.

Among the early examples are the Synthesizer Generator [45], mentioned above, as well as the Meta Environment [27]. The latter 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 [28] and Spoofax [25]. 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 [15] also based on SDF, which supports library based language extension. Spoofax-based IDE support is discussed in [14].

SmartTools [2] 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 and 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 [18]. SmartTools provides support for much of the infrastructure and makes using Visitors simple.

For transformation, SmartTools provides Xpp, a transformation language that provides a more concise syntax for XSLT-based XML transformations.

LISA [36] (mentioned earlier) supports the definition of language syntax and semantics (via attribute grammars) in one integrated specification language. It 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. The use of this approach for incremental language development is detailed in [37]. However, users have to make sure manually that sub-grammars remain unambiguous with respect to the base grammar. The same is true for the combination of independently developed grammars. LISA supports interactive debugging and program state visualization based on interpreting programs based on the semantic parts of the language specification.

Eclipse Xtext[19] generates sophisticated text editors from an EBNF-like language specification. Syntactic composition is limited since Xtext is based on ANTLR [41] which is a two phase LL(k) parser. It is possible for a language to 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[20], so it can be used together with any EMF-based model transformation and code generation tool (examples include Xpand, ATL, and Acceleo, all located at the Eclipse Modeling site[21]). Static semantics is based on constraints written in Java or on third-party frameworks that support declarative description of type systems such as Xtext Typesystem[22] or XSemantics[23]. Xtext comes with Xbase, an expression language that can be used as the base language for custom DSL. Xbase also comes with an interpreter and compiler framework that makes creating type systems, interpreters and compilers for DSLs that extend Xbase relatively simple.

The Helvetia system [44] by Renggli et al. supports language extension of Smalltalk with an approach where 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 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 to be used with another host language. This is particularly true for IDE support, where the Smalltalk IDE is extended using this IDE's APIs. The approach discussed in this paper does not have these limitations: MPS provides a language-agnostic framework for language and IDE

[19] http://eclipse.org/Xtext
[20] http://eclipse.org/emf
[21] http://eclipse.org/modeling
[22] http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/
[23] http://xsemantics.sourceforge.net/

extension that can be used with any language, once the language is implemented in MPS.

Cedalion [46] 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 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 composed syntaxes. MPS starts from external DSLs and adds modularization, and, as a consequence of implementing base languages with the same tool, optional tight integration with general purpose host languages.

For a general overview of language workbenches, please refer to the Language Workbench Competition[24]. Participating tools have implemented a common example language and document the implementation. 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

In this paper we classify language composition approaches based on syntactic mixing and language dependencies. Other classifications have been proposed, for example by Mernik et al. [35]. Their classification includes Extension (concepts are added to a language, similar to Extension as defined in this paper) and Restriction (concepts are removed from a language). The latter can actually be seen as a form of Extension: to restrict a language, we create 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. We do not discuss Pipelining in this paper, because it does not *compose* languages, it just chain their transformations. Piggybacking refers to a language reusing concepts from an existing language. This corresponds to Extension with embedding flavor. In [13], Erdweg et al. also propose a classification. Extension is the same as in our paper. They also consider Restriction as a form of Extension, where the extension restricts the use of certain language concepts. They call Unification what we call Embedding: two independent languages are used together in the same fragment. The two languages are combined without an invasive change to either of them. Each of the languages may have to be extended to "interface" with the other one. They also introduce the term Self-Extension, which describes the case where extensions are developed by means of the base language itself, an approach which is used by internal DSLs (see below) and is beyond the scope of this paper.

We already discussed the language modularization and composition approaches proposed by Mernik et al. [35] in Section 1.4. In the Helvetia paper [44] 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

[24] http://languageworkbenches.net

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 notion 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 [49]. 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 base language code, and this mapping is expressed with special host language constructs instead of a separate transformation language. Macro systems 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 [3] where each macro has to begin with a unique name, and only a limited set of syntactic shapes is supported. In OpenJava [51], the locations where macros can be added is limited. More fine-grained Extensions, such as adding a new operator, are not possible. SugarJ, discussed above, can be seen as a sophisticated macro system that avoids these limitations.

A particular advantage of projectional editing is that it can combine several notational styles in one fragment; examples include text, tables, symbols (fraction bars, square roots or big sums). All of these notations are seamlessly integrated in one fragment and can be *defined with the same formalism, as part of the same language* (as mentioned earlier, MPS supports text, tables and syntax. Graphics will supported in 2013). Other approaches for integrating different notational styles exist. For example, Engelen et al. [12] discuss integrating textual and graphical notations based on grammar-based and Eclipse modeling technologies. However, such an approach requires dealing with *separate* tools for the graphical and the textual aspects, leading to a high degree of accidental complexity in the resulting implementation and mismatches in the resulting tool, as the author knows from personal experience.

Internal DSLs are languages whose programs reside *within* programs expressed in a general purpose host language. In contrast to the Embedding approach discussed in this paper, the DSL syntax and semantics are also defined with this same host language (as explained by Martin Fowler in his DSL book [17]). 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, Hofer et al. describes internal DSLs in Scala [23]. The landmark work of Hudak [24] introduces internal DSLs as

language extensions of Haskell. While Haskell provides advanced concepts that enable creating such DSLs, they 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 built with static languages such as Scala suffer from these limitations. Since we consider IDE modularization and composition essential, we do not address internal DSLs in this paper.

7 Summary

MPS is powerful environment for language engineering, in particular where modular language and IDE composition is concerned. We have seen in this paper how the challenges of composing the concrete syntax are solved by MPS and how it is also capable of addressing modularity and composition of type systems and generators. Code completion, syntax highlighting and error marks for composed languages are provided automatically in MPS. The major drawback of MPS is its non-trivial learning curve. Because it works so different from traditional language engineering environments, and because it addresses so many aspects of languages (incl. type systems, data flow and refactorings) mastering the tool takes a significant investment in terms of time: experience shows that ca. 4 weeks are necessary. 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. J. Arnoldus, J. Bijpost, and M. van den Brand. Repleo: a syntax-safe template engine. In C. Consel and J. L. Lawall, editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 25–32, Salzburg, Austria, 2007. ACM.
2. 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, Part of ETAPS 2001, Proceedings*, volume 2027 of *Lecture Notes in Computer Science*, pages 355–360. Springer, 2001.
3. 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.
4. 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.
5. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 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 J. M. Vlissides and D. C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, pages 365–383, Vancouver, BC, Canada, 2004. ACM.
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, 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. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf, 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. L. Engelen and M. van den Brand. Integrating Textual and Graphical Modelling Languages. *Electronic Notes in Theoretical Computer Science*, 253(7):105–120, 2010.
13. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear.
14. S. Erdweg, L. C. L. Kats, C. Kastner, K. Ostermann, and E. Visser. Growing a Language Environment with Editor Libraries. In E. Denney and U. P. Schultz, editors, *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE 2011)*, pages 167–176, New York, NY, USA, 2011. ACM.
15. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 391–406, New York, NY, USA, 2011. ACM.
16. M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, 2005.
17. M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
19. D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, Volume 37(10):64–72, 2004.
20. L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information & Software Technology*, 46(7):465–472, 2004.
21. G. Hedin and E. Magnusson. JastAdd–an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
22. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN*, 24(11):43–75, 1989.
23. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, *Generative Programming and*

Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings, pages 137–148. ACM, 2008.
24. P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse, ICSR '98*, Washington, DC, USA, jun 1998. IEEE Computer Society.
25. 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).
26. L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. 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 918–932, Reno/Tahoe, Nevada, 2010. ACM.
27. P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
28. 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.
29. 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.
30. B. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
31. M. Bravenboer and E. Dolstra and E. Visser. Preventing injection attacks with syntax embeddings. In C. Consel and J. L. Lawall, editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 3–12, Salzburg, Austria, 2007. ACM.
32. 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, Proceedings*, volume 6823 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2011.
33. R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, 7(5):472–482, 1981.
34. 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.
35. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
36. 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, Part of ETAPS 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2002.

37. M. Mernik and V. Zumer. Incremental programming language development. *Computer Languages, Systems & Structures*, 31(1):1–16, 2005.
38. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.
39. D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2):91–105, 1985.
40. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003, Part of ETAPS 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2003.
41. T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
42. S. W. Porter. Design of a syntax directed editor for psdl. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1988.
43. 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.
44. 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.
45. T. W. Reps and T. Teitelbaum. The Synthesizer Generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48, New York, USA, 1984. ACM.
46. B. Rosenan. Designing language-oriented programming languages. In *SPLASH '10: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, New York, NY, USA, 2010. ACM.
47. 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.
48. 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.
49. G. L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.
50. C. A. Szyperski. *Component software - beyond object-oriented programming*. Addison-Wesley-Longman, 1998.
51. M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-Based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering, Papers from OORaSE 1999, 1st OOPSLA Workshop on Reflection and Software Engineering, Denver, CO, USA, November 1999*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 1999.
52. M. Voelter. Implementing Feature Variability for Models and Code with Projectional Language Workbenches. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development 2010*.

53. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Systems, Programming, Languages and Applications: Software for Humanity, SPLASH/Wavefront*, 2012.
54. M. Voelter and E. Visser. Product Line Engineering using Domain-Specific Languages. In E. S. de Almeida and T. Kishi, editors, *15th International Software Product Line Conference (SPLC), 2011*, pages 70–79. CPS, 2011.
55. H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *SPE*, 38(10):1073–1103, 2008.
56. 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.
57. E. V. Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. 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, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2002.
58. E. V. Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute Grammar-Based Language Extensions for Java. In E. Ernst, editor, *ECOOP 2007 - 21st European Conference on Object-Oriented Programming, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599. Springer, 2007.