Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS

Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS

Markus Voelter[1], Konstantin Solomatov[2]

[1] independent/itemis, Oetztaler Strasse 38, 70327 Stuttgart, Germany
[2] JetBrains s.r.o. Na Lysinach 443/57, Prague 4, 147 00, Czech Republic
voelter@acm.org, solomatov@gmail.com

Abstract. The language community has had a discussion about different styles of languages for a long time: big languages with many specific concepts (ABAP, Cobol), or small languages with few, but very powerful and orthogonal concepts (Lisp, Smalltalk)? With the advent of projectional language workbenches a new class of language becomes possible that can combine the best of both approaches: modular languages. Modular Languages use a relatively small general-purpose core and can be extended with more (domain specific) concepts as needed. Projectional language workbenches support this approach by alleviating the problems of language composition for parser-based languages by not using grammars and parsers at all. They also enable IDE extension as a side benefit. In this paper we argue why modular languages are useful and illustrate the idea with a couple of small examples. We also present a number of language composition techniques for JetBrains MPS, an open source projectional language workbench.

Keywords: projectional editing, language workbenches, language composition, domain-specific languages, modular languages

1 Introduction

There are various flavors and kinds of programming languages proposed or in use today. One way of grouping them is into big and small. Big languages have a large number of constructs, each with a very specific purpose (think: Cobol or ABAP). Small languages have a small set of language constructs, those being very powerful, orthogonal, and easily combinable (think: Lisp or Smalltalk). This paper advocates another alternative: modular languages. Modular Languages use a relatively small general-purpose core and can be extended with more (domain specific) concepts as needed. This idea is not necessarily new [7, 18, 19]. However, traditionally, modular languages have been hard to achieve because of the inherent problems in grammar composition and modular IDE construction [1, 20]. Projectional language workbenches [2] do away with these problems and make modular languages and IDEs easily possible.

A modular language consists of a minimal language core, plus a library of language modules that can be imported and used in a program as needed based on the task at hand. Modules have clearly defined dependencies, and elements from one language module can reference elements from another language module. Code written in one module's language can be embedded into code written in other modules' languages. For example, one could embed Java in XML to get XML templates or one could embed SQL in Java to handle database access. As a consequence of how projectional editors work, the full range of IDE services (syntax highlighting, code completion, or static error checking) are available for all programs, even if they are embedded in programs written in other languages. Since powerful IDEs are mainstream today and often a deciding factor for language adoption, this is a very important concern.

Each language module addresses a specific concern of software development in general, of a specific system or platform or of a specific business domain. A language module is a little bit like a traditional framework or library, but it comes with its own syntax, IDE support, type system, and compiler or transformation engine. Once a language module is imported, it's concepts behave as an integral part of the program, i.e. the user does not notice a difference (regarding symbolic integration and IDE services) between the various language modules. To illustrate the idea, figure 13 shows SQL embedded into Java and figure 16 shows a language module for robot control embedded in a C program.

Modular languages are *flexible* because users can use only those language concepts in a program they really need. They are *extendable* because a module can be custom built. Modular languages are *productive* because the scope of applicability of the module can vary from general-purpose to domain-specific. They are also *simple to use* because users don't have to learn (or expressly ignore parts of) huge monolithic languages, they only have to deal with those language modules they really use. Modular languages are *convenient* because, if the right tools are used, they come with their own IDE just like traditional languages.

The core contribution of this paper is to illustrate the how language modularization and composition can be achieved with projectional language workbenches and to show that projectional editors are mature and usable enough for practical use (see chapter 7).

The rest of the paper is structured as follows: chapter 2 introduces the idea of modular languages and argues why they are a worthwhile goal. Chapter 3 introduces language workbenches, projectional editors and JetBrains MPS. Chapter 4 looks at related work in language composition, incl. projectional and parser-based approaches. In chapter 5 we define an example language with MPS to illustrate projectional editing and language definition further. Chapter 6 illustrates language composition mechanisms. Chapter 7 briefly describes two real-world use cases where MPS is currently being used, and Chapter 8 provides a short summary.

2 A Library of Language Modules

An important argument for using domain specific languages and code generation is that languages and generators can be built that are closely aligned with a business domain or a specific concern of a software system (such as persistent data definition, workflow or component structures). The importance of the ability to define one's own languages and generators varies depending on the particular concern.

For a DSL that supports an insurance company's specific way of designing insurance contracts it is essential that the language is exactly aligned with the specifics of this company. This means the language has to be custom-built by the insurance company. Intentional Software have published a nice example for this case [3]. There are other similar examples: building DSLs to describe radio astronomy observations or describing data processing chains in particle accelerators or using hardware platform-specific DSLs in embedded software.

However, for a large range of concerns that are more software engineering-oriented, the abstractions are well-known, many of them being related to software architecture, framework use, or middleware. They can be put into module libraries and can be used directly, or extended slightly to tailor them to a specific architecture. Here are some examples.

- Hierarchical components, ports, component instances, and connectors. These abstractions are the backbone of many architecture description languages [4,5,6] and hence are a safe bet for reuse as a language module.
- Definition of rich contracts, i.e. interfaces, pre- and post conditions and protocol state machines
- Data structure definition according to the relational model or hierarchical data as in XML and XML schema, including descriptions for persisting, querying and navigating the data
- Various paradigms of concurrency such as shared memory and locking, actors, transactional memory, or data flow.
- Various paradigms of (remote) communication, such as message passing, synchronous and asynchronous remote procedure calls, and service invocations

It is certainly not possible to define all these language modules completely independent of each other, a clear layer structure between the modules is necessary. The language modules have to be designed to work with each other. Interfaces on language level support "plugging in" new language constructs in well-defined places. The modules might for example rely on a minimal core language that comes with object orientation, an expression language and first-class support for functions, close to languages like Java, C#, Scala, Clojure or C# today. These core languages serve as the basis for extension through modules. The core of part of this paper explains techniques for language extension and composition based on projectional editors in general, and JetBrains MPS [9] specifically.

Many of the above mentioned architectural concerns interact with frameworks, platforms and middleware solutions. It is not useful to put the specifics of technology solutions into the language. It is crucial that the abstractions in the language modules remain independent of specific technologies. In addition, when interfacing with a specific technology, additional (hopefully declarative) specifications might be necessary. Such a technology mapping should not be part of the core program, but should be a separate specification that references elements from the core program. This is like plug-ins: a language module might define a language for specifying persistence, distribution, or contract definition. Based on an open generation infrastructure, different technology suppliers can support customized generators that map programs to the APIs defined by their technology, taking into account possible additional specifications that detail this technology mapping. This is similar to service provider interfaces (SPIs, [10]) in any of today's industry standards. For example, a persistence language could use SPIs to generate code for several different Java persistent frameworks such as Hibernate, EclipseLink or the Java Persistence API.

3 Projectional Language Workbenches

The term Language Workbench has been coined by Martin Fowler in 2005 [2]. In his article he defines it 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.

Note how points 2, 3 and 4 imply projectional editing. In the meantime, Martin Fowler, and the community as a whole uses the term language workbench also for tools that use (modern) parsing techniques. Since this paper specifically adresses projectional editing, we use the term *projectional* language workbench.

The most important characteristic of projectional editors is that all text, symbols, and graphics are projected. Projectional editing is well-known from graphical modeling tools (UML, ER, State Charts). The model is stored independent of its concrete syntax, only the model structure is persisted, often using XML or a database. For editing purposes this abstract syntax is projected using graphical shapes. Users use mouse gestures and keyboard actions tailored to graphical editing to modify the abstract model structure directly. While the concrete syntax of the model does not have to be stored because it is specified as part of 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 interaction patterns and gestures known from "real text" to work. For a projectional editor to be useful, it has to "simulate" interaction patterns known from real text.

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 easily possible, because languages composition cannot result in ambiguous grammars [1, 19, 20].
- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions.
- Because projectional languages by definition need an IDE for editing (it has to do the projection!), language definition and extension always implies IDE definition and extension. The IDE will provide code completion, error checking and syntax highlighting for all languages, even when they are combined.
- Because the model is stored independent of its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints [11] of the overall program can be stored in one model, but editing can still be viewpoint specific. It is also possible to store out-of-band data, i.e. annotations on the core model/program. Examples of this include documentation, pointers to requirements (traceability) [12] or feature dependencies in the context of product lines [13].

As a side effect, language workbenches deliver on the promise of removing the difference between what is traditionally called programming and what is traditionally called modeling. This distinction is arbitrary anyway: as software developers we want to express different concerns of software systems with abstractions and notations suitable to that particular concern (graphical, textual, symbolic), formally enough for automatic processing or translation, and with good IDE support. Projectional language workbenches deliver on this goal.

JetBrains MPS

JetBrains' Meta Programming System [9] is an open source projectional language workbench, so all the statements made earlier apply to MPS. It comes with Java and XML as extendible default languages. Defining a language includes (a) defining the language concepts (abstract syntax), (b) defining the editor for the concepts and (c) defining a generator (compiler). For a language whose programs should be processed with a text-oriented tools (such as existing compilers or interpreters) the generator outputs text. For higher-level languages, the generator transforms the programs into programs expressed in lower level languages. In this case, the generators are not text generators, they transform between abstract syntax trees (this process is explained in more detail below).

Editing the tree as opposed to "real text" needs some getting 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 editing that resembles modern IDEs that use automatically expanding code templates. This makes editing quite convenient and productive in all but the most exceptional cases.

4 Related Tools and Approaches

MPS is not the only projectional workbench and projectional workbenches are not the only approach to language modularity and composition. For example, the Intentional Domain Workbench (IDW) [14] is another projectional editor that has been used in real projects. An impressive presentation about its capabilities can be found here [3]. IDW is conceptually very similar to MPS, although quite different in many details. We will not provide more details here, partly because Intentional Software is very sensitive about publishing information about details of their technology.

MetaEdit+ [15] is an example of a purely graphical language workbench (tables and forms can also be used). Languages are defined via meta models, constraints, code generators and graphical symbols associated with the meta-model. Language concepts can be represented by different symbols in different diagrams types and elements from several languages can be used in one diagram. Elements from language A can reference elements in language B. This is not surprising since graphical language workbenches are (and have always been) projectional. Since this paper focuses mostly on textual language, we include MetaEdit+ here only for completeness.

Eclipse Xtext [16] supports the creation of extremely powerful text editors (with code completion, error checking and syntax coloring) from an enhanced EBNF-like grammar definition. It also generates a meta-model that represents the abstract syntax of the grammar as well as a parser that parses sentences of the language and builds an instance of the meta-model. Since it uses Eclipse EMF [17] as the basis for its meta models, it can be used together with any EMF-based model transformation and code generation tool models (examples include Xpand, ATL, and Acceleo , all at [17]). It is easily possible to establish references between models expressed using different languages. Code completion for references into other models as well as cross-model and cross-language consistency checks in the editor are supported out of the box. Actual language composition is quite limited, though. It is possible to make a language extend one other language. Concepts from the base language can be used in the sub language and it is possible to redefine grammar rules defined in the base language. Creating new subtypes (in terms of the meta-model) of language elements in the base language is also possible. However, it is not possible to define different representations of the same element (except the concrete syntax of the reference) and it is not possible to reuse and embed arbitrary languages or language modules. This is mainly because the underlying parser technology is antlr [21] which is a classical two phase LL(*) parser which has problems[1] with grammar composition [1]. Single inheritance language extension is not enough in practice; it would be like object-oriented programming with only inheritance and no delegation or traits.

SDF [22], developed by the University of Delft, uses scannerless parsers. Consequently, languages can be embedded within each other. Code generators are implemented via term rewriting on the abstract syntax, but rendered in the concrete syntax! Currently SDF is mainly a set of command-line tools, but IDE support (with automatically generated eclipse editors) is in progress as part of Spoofax [23].

Monticore [24] is another parser based language engineering environment that generates parsers, meta-models, and editors based on extended grammar. Currently, the group at RWTH works on modularizing languages [25]. Languages can extend each other and can be embedded within each other. An important idea is the ability to not have to regenerate the parsers or any of the related tools after a combined language has been defined.

FURCAS [8] is a tool that is developed by SAP and the FZI Karlsruhe. FURCAS stores models in an abstract structure. However, for editing it "projects" the model into plain ASCII. So when editing the model, users actually edit ASCII text. Consequently, syntax definition also includes a definition of indentation and white space conventions, otherwise the projection could not work. Second, a lot of effort has to be put into a retaining object identity [26]. If an abstract structure is projected into text, and then for example something is moved around and saved back into the abstract structure, it has to be made sure the objects (identified by UUIDs) aren't deleted and re-created but really just moved. This is important to make sure that references between models which are based on the UUID's and not (qualified) names remain valid. By using scannerless parsers it is possible to combine different languages, however a combined grammar has to be defined and the parser has to be regenerated to be able to use the composed language. As a consequence of the

[1] In classical two-phase parsers, grammar combination is hard because recognition is not context aware. If two grammars define the same character sequence as different tokens, the combined grammar will be invalid. There are more subtle problems with as well, but this one is the most obvious. Scannerless parsers (such as SGLR parers [37] or parsing expression grammars [36]) don't have this problem, but have other challenges.

projectional approach is possible to define several syntaxes for the same abstract structure or define views and subsets for a model. FURCAS also generates IDE-like editors (based on Eclipse).

Another example of a grammar and parser based language composition system is LanGems [34, 35]. It supports role-based meta modeling and concrete syntax definition. However, if the combined syntax is ambiguous, manual disambiguation by invasively changing the syntax definition is required. While this may or may not be a problem in practice, projectional editors do not have this problem. Since there is no grammar, no ambiguities can arise; users have to disambiguate elements when they enter them into the program. Note that in case of MPS, no "combined language" has to be defined explicitly when several language modules should be used within one program. Users just import the language module.

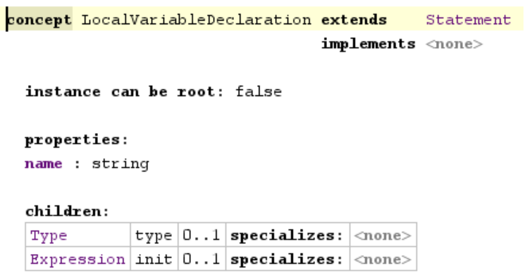
Note that while parser-based approach are becoming more flexible (as illustrated by some of the tools mentioned in this section), they will likely not be able to work with non-parseable code, inlined tables or diagrams or annotations. I describe some of these techniques below, since they are straight forward in projectional editors.

5 Defining an example language extension with MPS

This section shows the definition of a language with a projectional environment, JetBrains MPS. This is not intended as a tutorial for MPS. It is intended to provide an impression of how these kinds of tools work in principle.

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

This example illustrates the construction of a local variable declaration statement (like *int i = 2*3;*). We will integrate this new statement into a procedural language that has statements and expressions, such as Java or C#.

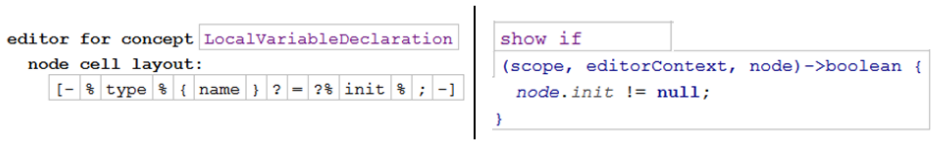


```
concept LocalVariableDeclaration extends    Statement
                                 implements <none>

  instance can be root: false

  properties:
  name : string

  children:
  Type       type 0..1 specializes: <none>
  Expression init 0..1 specializes: <none>
```

Figure 1: Language Structure definition for the local variable definition

Structure. Language definition starts out by defining the structure (abstract syntax) of a *LocalVariableDeclaration (*shown in figure 1). It extends *Statement*, so it can be used in *StatementLists*, such as the *body* of a *Procedure*. It contains a string property called *name* (the *i* in the example), a child of type *Type* (example: *int*) as well as an *Expression* called *init* (example: *2*3*). Note that *Statement, Expression* and *Type* are language concepts assumed to be defined in the language to which the *LocalVariableDeclaration* is added.

Concrete Syntax Definition. Instead of defining a grammar and deriving the abstract syntax, in MPS the editor definition is based on the language structure as shown in the left part of figure 2.



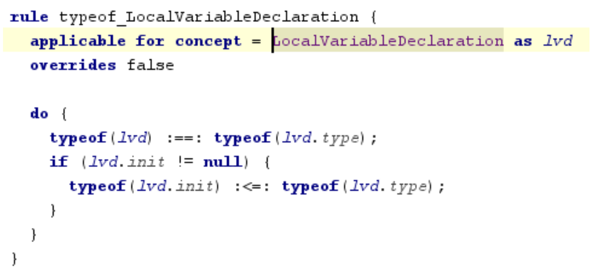
```
editor for concept LocalVariableDeclaration
  node cell layout:
    [- % type % { name } ? = ?% init % ; -]
```

```
show if
(scope, editorContext, node)->boolean {
  node.init != null;
}
```

Figure 2: Syntax/Editor definition, the condition for the *init* expression is on the right

The editor for the *LocalVariableDeclaration* consists of an indent-collection (*[- ... -]*), a kind of collection which results in a text-like presentation (horizontal, with line breaks). It contains the representation of the type (*%type%*), the name (*{name}*) and the init expression (*%init%*) if it has one (*?*). The condition (right half of figure 2) makes sure the initialization expression, as well as the equals sign, is only shown if an initialization expression is provided. Finally, the horizontal collection contains a semicolon - a simple constant. It is assumed that editors for *Type* and *Expression* are already defined in the base language. Those editors are simply used (i.e. embedded) here as we refer to *%type%* and *%init%*.

Type Systems. Type systems [27] defines and validate variable types. Every non-trivial language has a type system and MPS comes with a DSL to define type systems for languages.



```
rule typeof_LocalVariableDeclaration {
  applicable for concept = LocalVariableDeclaration as lvd
  overrides false

  do {
    typeof(lvd) :==: typeof(lvd.type);
    if (lvd.init != null) {
      typeof(lvd.init) :<=: typeof(lvd.type);
    }
  }
}
```

Figure 3: Definition of the type system rules

The type system rules in Figure 3 define that the type of the *type* property and the type of the overall statement have to correspond. In other words, the type of the whole statement is the type given in its *type* property. The second rule (within the *if* block) says that if an *init* expression is defined, the type of *init* must be a subtype of the type of the defined variable. Type equations not shown here establish the types of the *Type* and *Expression* concepts.

Editor Tuning. At this point the definition of the language and the basic editor, as well as the type system are finished. However, if you'd try to use the new *LocalVariableDeclaration* statement, you'd have to literally press control space in the editor, select the concept *LocalVariableDeclaration* and use tab or the mouse to fill in the various properties (*type*, *name*, *init*). This is unacceptable. Additional steps have to be taken to allow users to simply type *int i = 2*3;* the way they're used to.

The first step is to make sure users can select a *Type* in a *Statement* context instead of the *LocalVariableDeclaration*. A wrapper is defined that says: "Whenever a *Statement* is expected, users can also enter a *Type*. Once the *Type* is entered, replace the *Type* node with a new instance of *LocalVariableDeclaration* and put the original *Type* into its *type* field. Then move the cursor to the *name* field". MPS comes with specific support for these kinds of transformations, so implementing this wrapper can be done in a minute (literally).

The next step is to detect an equals sign in the variable name. If we find one, we need to remove it from the name and add a new *Expression* node into the *init* field of the *LocalVariableDeclaration*, and move the cursor there. Effectively this means that we can just type equals as part of the variable name and then continue with the initialization expression, as expected.

Finally, it must be possible to type expressions linearly. *2*3* is a tree with the root *** and the two children *2* and *3.* Without any customization, users would first have to enter the * and then the two arguments. An in-place transformation enables linear entering of the expression: if within an *IntConstantExpression* users type an *, a new *MultiplicationExpression* is created, the original *IntConstantExpression* is replaced with the *MultiplicationExpression* and the original *IntConstantExpression* is put into the left argument of the *MultiplicationExpression.* Finally the cursor is moved into the right expression to accept another *IntConstantExpression*.

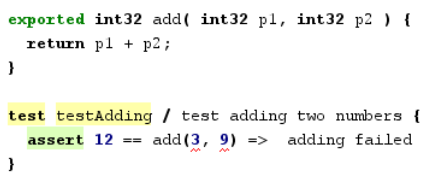
We explain all of this to illustrate what's involved in defining languages in a projectional editing environment. As the examples below will show, projectional editing has a number of intriguing advantages, but there is also a price to pay: this price is the additional effort that goes into defining a nicely usable editor. The tools (MPS in this example) come with the necessary facilities and makes defining these editor convenience functions relatively easy.

Generation and Compilation. In the *int i = 2*3;* example above code generation is relatively simple since the target language probably supports local variables in this way. So the generator simply renders a text for the tree structure. However, in general, domain-specific languages, or domain-specific extensions of general purpose languages cannot be directly executed. The model has to be translated into a language for which some kind of execution infrastructure (a compiler or interpreter) exists.

In an environment where models and programs are treated the same in that they are both stored as an abstract syntax tree and projected for editing, there are two different scenarios for code generation:

- domain-specific languages or language extensions typically need to be mapped to general purpose languages such as Java or C.
- Since the general purpose languages themselves are represented via an AST and projection, the programs cannot be feed directly to the compiler, a text representation has to be generated from them.

The second case is rather straight forward. It is basically pretty printing (aka unparsing), since the syntax of the language (abstract + concrete) directly resembles the text to be generated. The first case is more interesting. Since both, the DSL (or extension) and the target language are represented as AST + projection, such a code generator is in fact a model transformation in the sense that the DSL AST is transformed into the target language AST. There is no actual text generation.



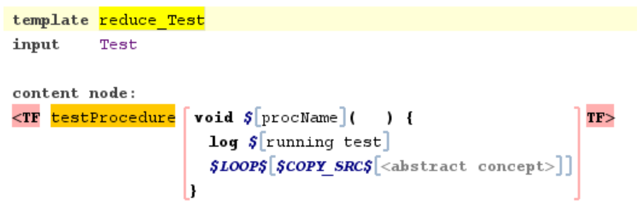
```
exported int32 add( int32 p1, int32 p2 ) {
  return p1 + p2;
}

test testAdding / test adding two numbers {
  assert 12 == add(3, 9) =>  adding failed
}
```

Figure 4: Example code for ilustrating code generators

As the example in figure 4 illustrates, this has one very intriguing advantage. The code contains a procedure *add* that adds two *int32*, as well as a test case in a C-like language. Test cases are not part of C, so they constitute an extension of C.

For compilation, these test cases have to be mapped to something that is basic C, to make sure the C-to-text mapping can translate it to text for compilation. Figure 5 shows a reduction rule that maps a unit test to a procedure.

A *procedure* is generated from the *Test* (see *input* field). It contains a *log* statement which outputs the name of the currently running test into some kind of log. The generator then iterates over all the statements in the test's body and reduces them in turn using their reduction rules. Note that there is a little bit more to these templates that is not shown in figure 5 such as expressions that calculate the name of the to-be-created procedure as well as an expression that tells the *LOOP* to iterate over the statements in the test's body.



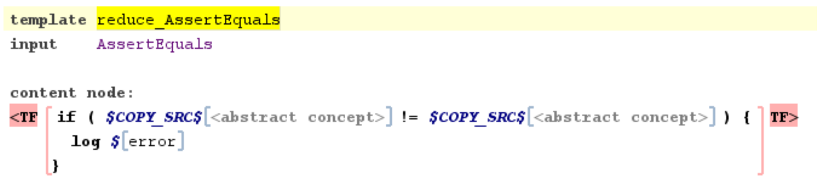
```
template reduce_Test
input    Test

content node:
<TF testProcedure [ void $[procName](    ) {                  ] TF>
                      log $[running test]
                      $LOOP$[$COPY_SRC$[<abstract concept>]]
                    }
```

Figure 5: A reduction rule for test cases

If a statement in the body is not plain C, i.e. if it is a domain-specific extension, then this statement must have a reduction rule itself to map it to plain C. This reduction rule is automatically invoked. Figure 6 shows the reduction rule for the *assert* statement, which is the only extension to C we use in the body of test cases.

This reduction rule maps an *AssertEquals* to an *if* statement that compares the actual with the expected expressions in the *AssertEquals*. If the comparison fails, the *log* statement is used to output an error message reporting the failed test case.

Here is the point of this example: Both reduction rules use the *concrete syntax of the target language* in the templates, so it looks as if this is a text-generation-template, when in fact it is a model transformation. To make this possible, the existing projections for the target language are used to project the resulting model in the syntax of the target language. This is visually similar to text generation templates that also use the concrete textual syntax of the target language. But the projectional approach goes further: when writing the to-be-generated code, the IDE provides syntax coloring, code completion and error checking for the template code. This is extremely hard to do for parsed languages, because the parser of the target language has to be embedded into the template language, and IDE services (code completion, syntax coloring, static checks) would have to be merged as well. We have not yet seen such as tool in parser-based languages, although the work of the Spoofax [23] team is very promising.



```
template reduce_AssertEquals
input    AssertEquals

content node:
<TF [ if ( $COPY_SRC$[<abstract concept>] != $COPY_SRC$[<abstract concept>] ) { ] TF>
  log $[error]
}
```

Figure 6: The reduction rule for assert statements

5 Language Composition and Extension

Language construction and composition in MPS is very much like object-oriented programming, the aforementioned problems of parser/grammar composition do not exist. This section illustrates a couple of approaches. Note that for all of the extension and reuse mechanisms explained below, no access to the source of the reused or extended languages is necessary. They could be available in binary for (as a JAR file in MPS).

Inheritance. Languages can extend other languages. Figure 7 shows two languages *A* and *B* where *B* extends *A*. Sublanguages can define concepts which extend concepts defined in the base language. Often the extended concepts (*X* in the example) are *abstract* or are implemented as concept interfaces.

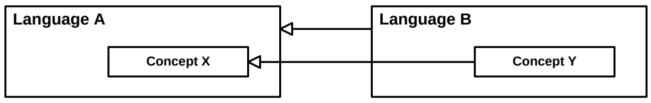


Figure 7: Language Inheritance

A program written using language *B* can use all the concepts from language *A* as well as those from language *B*. Subconcepts (such as *Y* in the diagram) can be used everywhere where *A* expects an instance of *X*. This is just like polymorphism in object oriented programming. For example, if language *A* defines the *Statement* concept, *B* could define a subtype *LockStatement* which then can be used wherever *A* expects a *Statement*.

Note that it is possible to define constraints that restrict the places where *Y* can be used in *B*. For example, if the sublanguage contains "concurrent procedures", the *yield* statement might be restricted to be used in the body of concurrent procedures.

It is also possible to define methods on language concepts. Specifically, abstract concepts can define abstract methods which can be implement in concrete subconcepts. These methods can be used anywhere where property access is supported, such as in generators and editors.

Interfaces to provide context. A Java class natively contains attributes and methods. Let's consider adding *properties* (like the ones provided in C#). Properties should be accessible in *DotExpressions*:

```
class SomeClass {
  property x: String;
  public void someMethod() { this.x = "hello"; } }
```

The language definition has to make sure the "existing Java" recognize properties as something that can go on the right side of a *DotExpression*. One way of implementing it is to have the existing Java *Methods* and *Attributes* implement an interface *IClassMember*. A *DotExpression* expects an *IClassMember* on the right side of the dot. *Properties* can now also implement this interface, making them available on the right side of the dot as well.

Overriding the Generator. If a language extends another language it is possible to override the base language generator (see figure 8). In a model where only language *A* is used, *Generator(X)* is executed during translation. Once language *B* is included, instances of *X* are translated via *Generator'(X)*. This facility nicely supports target platform-specific translation of language concepts.

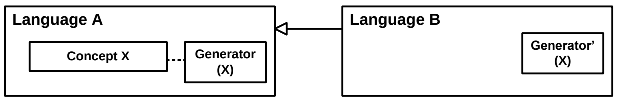
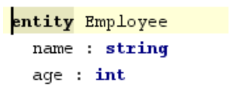


Figure 8: Overriding the generator in a sublanguage

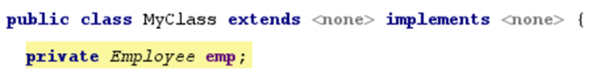
It is also possible to constrain the context in which the new translation rules should be applicable in a program: the same constructs can be translated differently depending on where they are used.



```
entity Employee
  name : string
  age : int
```

Figure 9: An example Entity

Adapters. Imagine extending Java by adding, in addition to classes, so-called *Entities*. Entities are like *structs* in that they have properties, but, for example, they could be persisted into a database. Figure 9 shows the definition of an *Entity*. To integrate them into Java, *Entities* should be types. That is, they should be usable as types in "regular" Java code, as shown in figure 10.



```
public class MyClass extends <none> implements <none> {
  private Employee emp;
```

Figure 10: An Entity used as the type of an attribute in a Java class

Figure 11 shows how this is achieved. In the left box, figure 11 shows how MPS' existing Java language provides a *Type* concept. Note how *ClassType* acts as an Adaptor [28] for *Class*es in the context of *Types*. The *ClassType* is-a *Type* and references the *Class* whose type it represents. To add *EntityTypes,* an *EntityType* is created which extends *Type* and points to the *Entity* whose type it should represent.

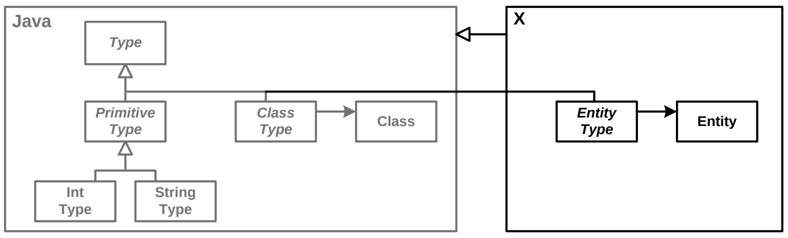


Figure 11: Existing Language Structure for Types and the extension for Entities

There's one more important detail to making this work. Consider the attribute declaration *private Employee emp;* again. Figure 12 shows the structure underlying it.

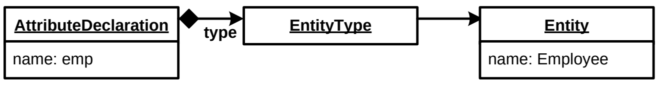


Figure 12: Instance structure for the attribute declaration

Imagine you're in the editor editing a member declaration and you want to enter the type. At this time you're likely to press *Ctrl-Space* to get a list of the existing types you can use here. However, you don't want the system to propose to *Create a new EntityType* (as suggested by the containment between *AttributeDeclaration* and *EntityType*); you really want it to show all the existing *Entities* for you to select from.

MPS provides this functionality by adhering to the following convention: if the concept structure expects an instance of a concept that has *only a single reference* feature (and no other features) then the following happens:

- execute the scope definition for *EntityType*'s single reference
- show the resulting elements (the visible *Entities*) in the code completion list
- upon selection, instantiate the concept (here: *EntityType*) to fill in current slot and set its single reference to the selected target.

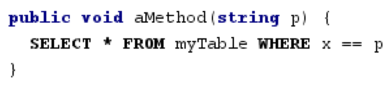
In some sense, MPS "jumps over" the intermediate adapter object. This approach supports the injection of adaptors into the structure without spoiling the editing experience.

Embedding independent languages. The above examples require that the specialized language explicitly extends the base language - in other words, when defining the specialized language, the base language had to be known.

However, there is also a use case where languages should be combined that have been developed independent of each other. As an example, consider embedding SQL into Java, where both SQL and Java have been implemented in MPS, but they have no relationship to each other. The generator for the Java language generates Java text for subsequence compilation. The generator for the SQL language also generates text that can be used to script a database system.

Let us look at the structure of two languages in the top part of Figure 14. Java contains *Methods* which own *MethodParameters*. A method also contains a list of *Statements* as its body. The *SimpleSQL* language contains *SQLStatements*, *SelectStatement* being a subclass. A *SelectStatement* contains a set of *SelectColQuery*s (*… where city=="Stuttgart"*). A *SelectColQuery* contains the column name (*city*) and a *ComparisonValue*. A *LiteralComparisonValue* represents a string literal (*"Stuttgart"*).

When integrating Java and SQL, *SQLStatements* should be usable in method bodies. Within an SQL query, method parameters should be usable as a comparison value. Figure 13 shows what the result looks like:



```
public void aMethod(string p) {
  SELECT * FROM myTable WHERE x == p
}
```

Figure 13: A select statement embedded in a Java method, referencing a method parameter

Note that the two languages are completely integrated, including code completion etc. for SQL, and the *p* in the where clause is actually a reference to the method parameter. Figure 14 shows how this is implemented: A new language *JavaWithSQL* extends both Java and SQL. Using the *Adaptor* pattern [28], a new concept *SQLStatementStatement* is defined which extends Java's *Statement* and contains an *SQLStatement*. This supports the embedding of *SQLStatements* in places where Java expects a Java *Statement* (i.e. in the method body). The same adaptor approach is used for the comparison value. The *MethodParamComparisonValue* extends SQL's *ComparisonValue* and points to a *MethodParameter* from Java. The resulting language provides code completion for the select statement as well as for the parameters.

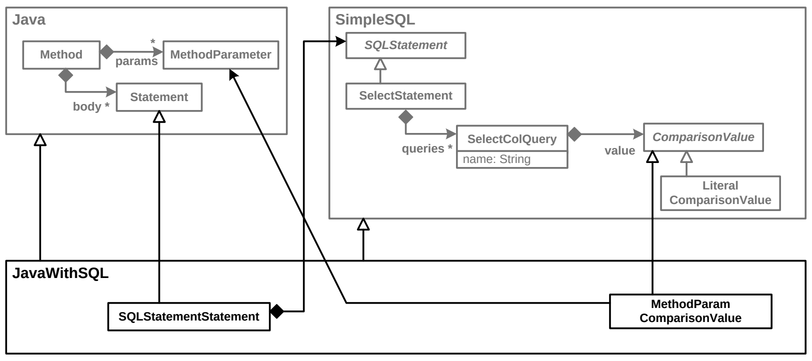


Figure 14: Structure of the integration between Java and SQL

Of course the two languages also have to be integrated semantically. In other words, the generated code for the integrated languages must fit together. To this end, the *JavaWithSQL* language specializes the transformations from *SimpleSQL* to make sure the result looks like the following:

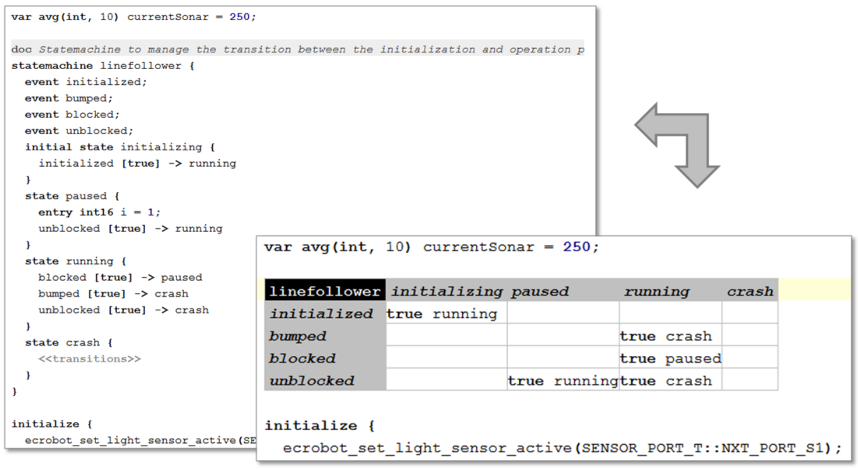
```
public void aMethod(String p) {
  StringBuffer sb = new StringBuffer("SELECT * FROM myTable WHERE ");
  sb.append("x" + "== \"" + sanitizeInput(p) + "\"");
  SomeSQLFramework.execute( sb.toString() );
}
```

While the projectional approach neatly solves all the syntactic problems, the approach is limited by semantic interactions of the generated low-level code. In the example above, it is possible to embed the SQL in the resulting (plain) Java code by wrapping it in a string. However, imagine the following situation:

- A language extension of Java called *XLanguage* has a language concept called *X*. In the implementing generator, the Java classes that embed *X* are made to inherit from a base class *XUtilities*.
- A language extension of Java called *YLanguage* has a language concept called *Y*. In the implementing generator, the Java classes that embed *Y* are made to inherit from a base class *YUtilities*.

For a class that embeds an X and a Y (which is no problem from a syntactic point of view), the resulting code will not work because Java classes can only inherit from one base class. It could be argued that the generators of *XLanguage* and *YLanguage* are badly designed (and I would agree!), but as a language developer you don't have control about how other, independently developed languages are implemented.

Different Notations. The ability to have several different notations for the same language concepts is not technically related to language extension. But it is nonetheless very useful. For example, in figure 15, a state machine can be represented as a table inline in the textual C program. The two notations can be switched back and forth.



```
var avg(int, 10) currentSonar = 250;

doc Statemachine to manage the transition between the initialization and operation p
statemachine linefollower {
  event initialized;
  event bumped;
  event blocked;
  event unblocked;
  initial state initializing {
    initialized [true] -> running
  }
  state paused {
    entry int16 i = 1;
    unblocked [true] -> running
  }
  state running {
    blocked [true] -> paused
    bumped [true] -> crash
    unblocked [true] -> crash
  }
  state crash {
    <<transitions>>
  }
}

initialize {
  ecrobot_set_light_sensor_active(SE
```

```
var avg(int, 10) currentSonar = 250;
```

linefollower	initializing	paused	running	crash
initialized	true running			
bumped			true crash	
blocked			true paused	
unblocked		true running	true crash	

```
initialize {
  ecrobot_set_light_sensor_active(SENSOR_PORT_T::NXT_PORT_S1);
```

Figure 15: Representing a state machine textually or as an embedded table

Language Annotations. Language annotations are a way of extending languages without invasively modifying them. It is possible for a language B to contribute additional properties to elements of language A. The annotation also defines how the additional property is rendered in the editor of A. This way, arbitrary "out of band" information such as documentation, requirements traces [12] or variability information [13] can be stored in the model. The usefulness of this feature for product line engineering is described in [33].

7 Examples

This section contains two examples where the modular language approach is being used. One covers web development and the other one addresses embedded systems.

Web Application Development. JetBrains, the developer of MPS has recently published a new product, the YouTrack issue tracking system (code named Charisma during its development). YouTrack is an interactive web application with many UI features known from desktop applications. YouTrack is the first product that was developed completely with MPS. 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. As a consequence of MPS' architecture as described in this paper, product-specifics can be added to these languages easily making sure the spent effort can be exploited in future products.
Web development involves many languages. In the browser, HTML, CSS, JavaScript and SVG are used. All these languages embed one another. On the Java-based server side, a set of descriptive languages is used, together with query languages (EQL, HBQL, SQL), template languages (JSP, Velocity) and of course the Java programming language at the core. JetBrains decided to wrap these platform specific implementation languages with a set of Java language extensions. For the sake of the example, we focus on describing the extensions used in the Java-based backend. The most notable of the languages used in YouTrack are *dnq* and *webr*.

dnq is a Java language extension for working with persistent data and queries. Almost all modern web applications store data in a database. However, database manipulation isn't very well supported in general purpose imperative languages such as Java. Developers use object relational mapping frameworks such as Hibernate, JDO or JPA so alleviate this problem. These frameworks basically map database rows to Java classes. However, because authors of these frameworks cannot change Java language, the integration is limited, hampering developer productivity. For example,

- Entity relations which are inherently bidirectional can't be easily expressed in Java. Consider a program which models organizational structure, consisting of *Departments* and *Employees*. When an Employee is added to a Department, both, the references in *Employee* and *Department* must be updated consistently.
- Relational databases optimize queries very aggressively. In order to accomplish these optimizations, queries should be expressed in SQL. However, it's much more natural to use the programming language for querying database, especially if the query language were integrated with the host language and its type system. To enable this, the programming language must be extended with query constructs and these must be translated into SQL when the program is compiled.

JetBrains has built Java extensions for both of these problems with MPS. The *dnq* language supports the expression of relations in a more natural way: unidirectional and bidirectional relationships can be declared, distinguishing between composition and references. Programmers can access them in a way similar to accessing fields in Java. *dnq* also includes a collections language, a language which supports the manipulation of collections in a way similar to .NET's LINQ. For example, ít supports code such as the following:

```
aColl.where({it=> it.val < 20 && it.val > 10}).select({it=> it*it});
```

This code is more semantically meaningful (delcarative) than procedural collections manipulation code which allows MPS to optimize such queries to the database.

The *webr* language is used for request handling in web applications. In web frameworks this tasks is typically accomplished by controller classes and HTML templates. To configure HTTP request handling, frameworks often use XML based descriptors. In order to process the HTML templates, template engines are used. Examples include JSP, Velocity or FreeMarker. *webr* supports this through Java language extension. Its template language combines XML and Java, relatively similar to JSP at first glance. However, based on MPS' ability to extend languages, *webr* provides much more freedom of what templates can contain. For example, in many template engines, it's impossible to add new constructs to the template language. In JSP it is possible using extension tags but the XML based syntax is quite verbose. In *webr* templates, developers can choose whatever syntax they like by defining a suitable language extension. An example used in YouTrack is a UI components language that is not limited to XML syntax.

webr also provides first-class support for controllers. For example, controllers can declare actions and attach them directly to events of UI components. Parameters are specified in special entities called template controllers. *webr* is well integrated with *dnq*, so for example, it is possible to use a persistent entities as a parameter to a page. The database transaction is automatically managed during request processing.

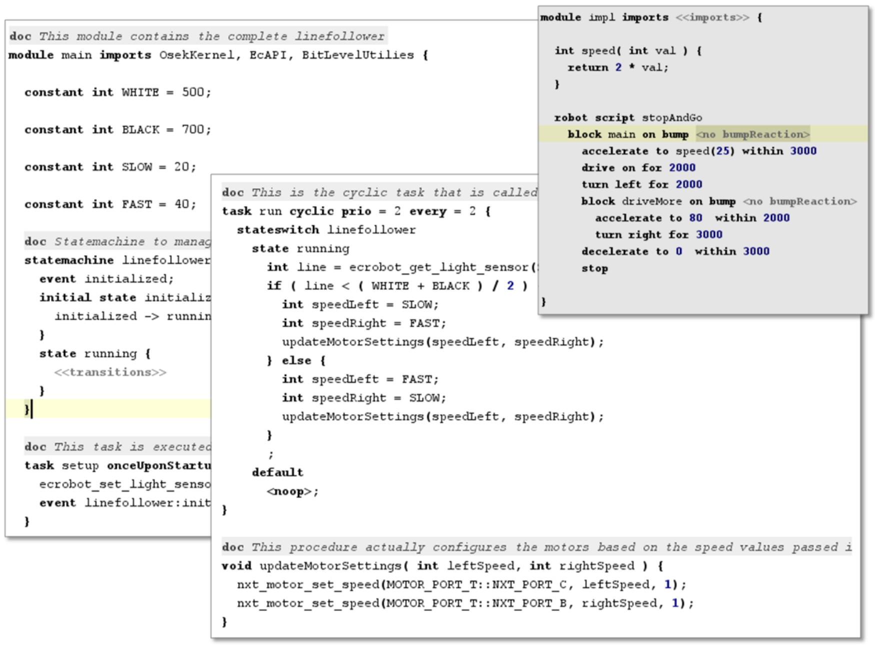
Implementation of the languages used for YouTrack took substantial effort but JetBrains gained many benefits from using MPS. The ability to change the language means developers don't have to write boilerplate code required by bare-bones J2EE. As a result, code is much easier to maintain. Experience shows that the language extensions are easier to learn than J2EE APIs. As an experiment, a student who had no experience in web development was tasked to create a simple accounting application. He was able to produce a web application with sophisticated Javascript UI in about 2 weeks!

It is instructive to look at the benefits of a language workbench in defining the *webr* and *dnq* languages. JetBrains has built many integrations of Java enterprise frameworks (such as Spring, Hibernate or Struts) into its IntelliJ IDEA Java IDE,. Experience shows that a qualified Java developer can create such an integration in about a year. In contrast, creating a language extension in MPS for an existing framework takes about one week. So, the productivity gain for tool developers is substantial.

Embedded Systems Development. Markus Voelter and Bernhard Merkle are currently working on a modular language for embedded development based on C. It is described in detail in another paper [29]. Here is a brief overview.

Embedded systems are becoming more and more software intensive. Consequently, software development plays an increasingly important part in embedded system development, and the software becomes bigger and more complex. Traditional embedded system development approaches use a variety of tools for various aspects of the system, making tool integration a major headache. 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 [30]), the proprietary and closed nature of modeling tools, the integration of models and code, traceability to requirements [12], long build times as well as the whole field of product line variability [13].

To address these issues, we propose to implement a modular modeling and programming language and tool that supports higher-level abstractions and system-specific extensions based on a projectional language workbench and to use code generation to C as a way of integrating with existing compilers and platforms. Some of the extensions could include a module system with visibility rules, physical quantities (as opposed to just *ints* and *floats*), first-class state machines, dataflow ports, mathematical notations, memory mapping and bit fields, as well as first-class support for various inter-process communication approaches (shared memory, message passing, bus communication).



```
doc This module contains the complete linefollower
module main imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;

  constant int BLACK = 700;

  constant int SLOW = 20;

  constant int FAST = 40;

  doc Statemachine to manag
  statemachine linefollower
    event initialized;
    initial state initializ
      initialized -> runnin
    }
    state running {
      <<transitions>>
    }
  }

  doc This task is executed
  task setup onceUponStartu
    ecrobot_set_light_senso
    event linefollower:init
  }
```

```
doc This is the cyclic task that is called
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      int line = ecrobot_get_light_sensor(
      if ( line < ( WHITE + BLACK ) / 2 )
        int speedLeft = SLOW;
        int speedRight = FAST;
        updateMotorSettings(speedLeft, speedRight);
      } else {
        int speedLeft = FAST;
        int speedRight = SLOW;
        updateMotorSettings(speedLeft, speedRight);
      }
      ;
    default
      <noop>;
}

doc This procedure actually configures the motors based on the speed values passed i
void updateMotorSettings( int leftSpeed, int rightSpeed ) {
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, leftSpeed, 1);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, rightSpeed, 1);
}
```

```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump <no bumpReaction>
      accelerate to speed(25) within 3000
      drive on for 2000
      turn left for 2000
      block driveMore on bump <no bumpReaction>
        accelerate to 80  within 2000
        turn right for 3000
      decelerate to 0  within 3000
      stop

}
```

Figure 16: Example code for the modular embedded DSL. The grey code on the top right is an example of the two-wheel-robot DSL.

As a proof of concept, we are currently building this modular embedded language based on MPS. We use Lego Mindstorms [31] as the target platform together with the Osek [32] operating system to ensure real-world relevance. The current showcase is a line follower robot. It uses a single light sensor to follow (one side of) a thick black line. It keeps track of the curving line by changing the speed of motors that drive the two wheels. The current state of the prototype contains language modules for components, tasks, state machines, bit-level data structures, physical quantities, documentation annotations, as well a core module with basically all of C. There is a clearly defined dependency structure between those languages, with the core language at the root. In addition, we have added a DSL for the simple control of the two-wheeled robot using commands such as *accelerate*, *turn left*, or *stop*. The implementation of this DSL is a model transformation down to the basic embedded languages: we generate tasks, procedures and state machines, which are then (automatically) transformed down to actual C and are compiled. Figure 16 shows example code.

As we continue to build Mindstorms applications with our language, it turns out that it is useful to extend plain C with embedded-specific concepts. Programs become more easily readable and can be analyzed more easily. It is feasible to package the various aspects into separate language modules and make incremental extension possible. It is also surprisingly little effort to build language extensions: developing the basic C implementation has taken us about 3 weeks. Adding the state machine facilities has been done in one afternoon. Creating the *robot routing* DSL on top was a matter of 4 hours, including the mapping down to tasks and state machines.

The next steps will be to add more abstractions to the language for large-scale embedded development, including a powerful component model, traceability to requirements and product line variability. We are also planning to build a real system (and not just Mindstorms examples) to verify the approach.

8 Summary

In this paper we have described how language composition works with MPS. Other projectional editors, such as the Intentional Domain Workbench, work differently in detail, but provide the same flexibility regarding language composition. Our experience in working with these tools makes us appreciate the flexibility and simplicity in language definition. Having worked with parser-based environments a lot, the projectional environments feel like a great relief and step forward. For example, Markus has tried a modular architecture description language with Eclipse Xtext before and gave up: being forced to implement the language modules via includes and preprocessors did not scale at all. The similar challenges faced in building the modular embedded DSL with MPS were trivial to solve.

While we are aware that parsing technology also moves forward, and language composition will become much more practical in these environments as well, we feel that projectional editors are a very promising way for the future, especially since they can integrate textual, symbolic and graphical notations. Traditionally, the biggest complaint about projectional (or structural) editing has been that the editing experience is not very convenient. As MPS demonstrates, these reservations are not valid (anymore).

Acknowledgements

We thank Iris Groher for the feedback on earlier versions of this paper.

References

1. Bravenboer, M., Visser, E.: Parse Table Composition, Separate Compilation and Binary Extensibility of Grammars, SLE'08 and http://swerl.tudelft.nl/bin/view/EelcoVisser
2. Fowler, M.: Language Workbenches - The Killer-App for Domain Specific Languages?, http://martinfowler.com/articles/languageWorkbench.html
3. Christerson, M., Kolk, H.: Domain Expert DSLs, http://www.infoq.com/news/2009/03/DSL-Magnus-Christerson-Henk-Kolk

4. University of California, xADL, http://www.isr.uci.edu/projects/xarchuci/
5. Carnegie Mellon University, ACME ADL, http://www-2.cs.cmu.edu/~acme/
6. Carnegie Mellon University, AADL, http://www.aadl.info/
7. Dmitriev, S., Language Oriented Programming: The Next Programming Paradigm, http://www.onboard.jetbrains.com/articles/04/10/lop/mps.pdf
8. Goldschmidt, T., Becker, S., Uhl, A., Textual Views in Model Driven Engineering 35th EUROMICRO Conference on Software Engineering and Advanced Applications
9. JetBrains, Meta Programming System, http://jetbrains.com/mps
10. Software Engineering Institute, Replaceable Components and the Service Provider Interface, http://www.sei.cmu.edu/library/abstracts/reports/02tn009.cfm
11. Wikipedia, View Model, http://en.wikipedia.org/wiki/View_model
12. Wikipedia, Requirements Traceability, http://en.wikipedia.org/wiki/Requirements_traceability
13. Software Engineering Institute, Software Product Lines,http://sei.cmu.edu/productlines/
14. Intentional Software, Intentional Domain Workbench, http://intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf
15. http://www.metacase.com
16. http://www.eclipse.org/Xtext/
17. http://eclipse.org/modeling/
18. Grimm, R.: Better extensibility through modular syntax. PLDI'06
19. Klint, P., L¨ammel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering Methodology 14(3) (2005) 331–380
20. Wyk, E.R.V., Schwerdfeger, A.C.: Context-aware scanning for parsing extensible languages, GPCE'07
21. http://www.antlr.org/
22. http://strategoxt.org/Sdf/WebHome
23. http://strategoxt.org/Spoofax
24. http://www.monticore.org/
25. Krahn, H., Rumpe, B., Völkel, S., MontiCore: Modular Development of Textual Domain Specific Languages TOOLS 2008
26. Thomas Goldschmidt, Towards an incremental update approach for concrete textual syntaxes for UUID-based model repositories, SLE 2008
27. http://en.wikipedia.org/wiki/Type_system
28. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
29. Voelter, M., Embedded Software Development with Projectional Language Workbenches, submitted to MODELS 2010, http://www.voelter.de/data/pub/Voelter-Models2010-EmbeddedSystemsDevelopmentWithProjectionalLanguageWorkbenches.pdf
30. MISRA Group, Misra-C, http://www.misra-c2.com/
31. Lego, Mindstorms, http://mindstorms.lego.com
32. Sourceforge.net, nxtOSEK, http://lejos-OSEK.sourceforge.net/
33. Voelter, M., Product Line Engineering with Projectional Language Workbenches, submitted to GPCE 2010, http://www.voelter.de/papers/ Voelter-ProductLineEngWithProjectionalLanguageWorkbenches.pdf
34. Wende, C., Thieme, N., Zschaler, S., A Role-based Approach Towards Modular Language Engineering, SLE 2009
35 .Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models, ECMDA-FA. (2009)
36. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation, POPL'04
37. van den Brand, M., Scheerder J., Vinju, J., Visser, E., Disambiguation Filters for Scannerless Generalized LR Parsers, LNCS, Volume 2304/2002