JetBrains MPS as a Tool for Extending Java

Vaclav Pech
JetBrains
vaclav@jetbrains.com

Alex Shatalin
JetBrains
alexander.shatalin@jetbrains.com

Markus Voelter
independent/Itemis
voelter@itemis.de

Abstract

JetBrains MPS is an integrated environment for language engineering. It allows language designers to define new programming languages, both general-purpose and domain-specific, either as standalone entities or as modular extensions of already existing ones. Since MPS leverages the concept of projectional editing, non-textual and non-parseable syntactic forms are possible, including tables or mathematical symbols. This tool paper introduces MPS and shows how its novel approach can be applied to Java development. Special attention will be paid to the ability to modularize and compose languages.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments - Programmer workbench

General Terms D.3.2 Extensible languages, D.3.4 Code generation, D.3.4 Translator writing systems and compiler generators

Keywords language extension, DSLs, development environments, formal methods

1. Introduction

JetBrains MPS is an open-source language workbench based on a projectional editor. It started as an experimental project about ten years ago with the aim to test and validate the ideas of Language Oriented Programming (LOP), as summarized in [2]. In essence, MPS attempted to bring code generation and projectional editing (also known as structured editing, see [12] for a good definition) to the programming mainstream, enabling easy language modularization and composition. The approach blurs the distinction between general-purpose (GPL) and domain-specific (DSL) programming languages with an intended positive impact on developer productivity. This paper introduces MPS, focusing on extensibility of languages and IDEs in general and for Java, specifically.

The term Language Workbench was coined by Martin Fowler [4]. Besides MPS, there are several other language workbenches including the Intentional Domain Workbench [8], MetaEdit+[1], Eclipse Xtext [2] and Spoofax [3] (for a brief comparison of some of these tools see Related Work in Section 4). For a general overview of language workbenches, please refer to the Language Workbench Competition[4].

Real-world use of MPS As a real-world proof of MPS, JetBrains has built YouTrack [5], a web-based bug tracking application. It heavily relies on an MPS-based set of languages for web application development and database mapping – Webr-DNQ. In less than three years YouTrack became an important part of JetBrains' product portfolio, validating the LOP idea as well as its implementation in MPS. More on how YouTrack benefits from LOP can be found in [9].

The MPS team cooperated with several universities to help them adopt MPS as a tool for language design research. For example, a student at the Faculty of Mathematics and Physics of Charles University in Prague implemented a real-time Java development environment as part of his master thesis in 2012 [3].

Although MPS is historically bound to Java and the JVM, its principles are universal and can be applied to any technological domain. The mbeddr project [11] successfully built a powerful development stack for embedded C development. They primarily target programmers who write C code for micro-controllers in robots, avionics or car appliances.

Further plans Inspired by the success of YouTrack and mbeddr, JetBrains has been gradually turning MPS into a solid open-source language workbench that is ready for the tasks in the mainstream of software development. The recent 2.5 version can be considered a fully functional, stable production-ready tool, targeting language and DSL designers as well as programmers. The open source license facilitates adoption for open-source and academic use.

2. Language Engineering in MPS

Eating its own dog food, MPS, like many other language workbenches, offers a set of DSLs for defining various aspects of languages. These include the *structure*, *editor*, *type system*, the *generator* as well as support for sophisticated IDE functionality, such as code completion, intentions, refactorings, debugger and dataflow analysis.

PPPJ'13,, September 11–13, 2013, Stuttgart, Germany..
Copyright is held by the owner/author(s).
ACM 978-1-4503-2111-2/13/09.
http://dx.doi.org/10.1145/2500828.2500846

[1] http://metacase.com

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

[3] http://spoofax.org

[4] http://languageworkbenches.net

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

You may also check out two introductory screen-casts on projectional editing in MPS [6] and on language definition[7], to get a concrete example of the discussed content.

Abstract Syntax As a first step when defining a new language we specify the *structure* (aka abstract syntax or meta model). This resembles object-oriented programming, language *concepts* are similar to classes. Concepts in MPS represent types of AST nodes and define the properties, methods, children and references that instance nodes may have. MPS offers three language aspects to define abstract syntax: the *structure* aspect defines the concepts, their properties and relationships, *constraints* restrict the allowed set of values for properties and references, and the *behavior* associates methods with concepts.

Concrete Syntax The second step is defining the editor for the concepts. This reflects the projectional nature of MPS. Since code in MPS is never represented as plain text (neither on the screen nor on disk), MPS languages are never parsed and thus no grammar is required. This enables the use of non-parseable notations, such as tables or mathematical symbols. Instead of a parser, we define editors for language concepts – a visual representation for AST nodes. MPS' editor definition capabilities let the language designer define editors for her new language constructs as well as override editors of concepts from existing languages. Editors can be modularized to support reuse of visual syntax elements and multiple editors can be defined for a concept, allowing the programmer to choose the notation that fits best the task at hands.

The default MPS editor needs some getting used to for programmers familiar with text editing. Building code from predefined language concepts instead of from individual characters definitely feels different. However, MPS goes a long way towards the editing experience of text editors. The editor's behavior can be tuned by specifying many of its dynamic characteristics using, for example, node factories, side transformations, or replacement rules.

IDE Functionality Developers today expect the editor to instantly assist them with code completion, smart navigation, intentions, refactorings, hints and code analysis. Being able to quickly build an IDE for a language is one of the most noteworthy advantages of MPS. The editor offers many of its essential features out-of-the-box, the remainder can be implemented by the language designer through appropriate language aspects. For example, whenever the developer hits `Ctrl-Space`, the code-completion pop-up menu is automatically populated with all type-compatible nodes that are in scope. The language designer can customize the scoping rules of the language, which will be reflected in the content of the code-completion menu. Also, if the language designer defines a Dataflow aspect for her language, the user will then get unreachable parts of the code highlighted as errors.

MPS seamlessly integrates with several version control systems (VCS), such as Git and Subversion. Diff and merge is provided on the concrete syntax level, so that, without extra effort on the language designer's side, code can be properly maintained through the projectional editor.

The MPS debugger is also extensible, allowing languages to be debugged in MPS. The user can set breakpoints and inspect values in the high-level language (DSL), avoiding exposure to the actual execution technology used.

Type System The next step is the definition of the type system. For example, the type of a condition of an `IfStatement` must be `boolean`, or a type of a `return` statement must be compatible with the return type of the surrounding method. MPS comes with a type system engine that is capable of evaluating type system rules to assign types to language elements (type inference) and to verify correctness of types. Typing rules are specified as a set of equations, such as the following:

```
infer typeof(ifStatement.condition) :<=: <boolean>;
```

where the `:<=:` symbol specifies *equal type or a subtype*. The engine tries to assign values to the type variables so that all equations are satisfied. The declarative approach (based on unification[8]) enables easy extension of the type system by adding additional equations and types.

MPS also support checks that are not directly related to typing rules. A dedicated type of rules, called *checking rules*, may analyze the model and report warnings or errors whenever an incorrect construct is detected. Examples for checking rules in Java include detecting the repetitive use of the same string literal, assignment to a `final` variable, ignoring a return value and many others. Language designers may associate automated fixes with checking rules, which users can choose to execute in order to fix the problem.

Code generation Languages in MPS also define transformation rules to lower-level languages or to plain text. The generation process in MPS consists of two phases. Phase one uses a template-based model-to-model transformation engine to reduce the program code into the target language, based on *reduction rules* specified in the generator. The target language may be further reduced based on its own reduction rules, and so on. When no further model-to-model transformation is applicable to a model, the second phase uses text generators to convert that final model into regular program text that can be fed into a compiler.

A language generator aspect consists of two main building blocks. *Mapping Configurations* define which concepts are processed with which templates and in which context. *Templates* define the code in the target language that will replace the specified piece of the source concept(s). Templates work differently from normal text generation templates such as for example Xpand[9], Jet[10] or StringTemplate[11], since in MPS they do not define text output, but instead model-to-model transformations. The generator developer first writes a structurally correct example model using the target language and then uses so called *macros* to tie the example model to the actual input from which we generate. Both templates and macros are well known concepts from other tools, but in MPS they work on the AST level (see Fig. 1).

Another of key benefits of the MPS approach to language definition is that the editor is capable of providing full IDE support for the target language when defining the template. A generator template is just another piece of code that mixes several languages, as far as the editor is concerned. Language modularization and mixing is discussed in [10].

[6] http://tv.jetbrains.net/videocontent/your-first-date-with-jetbrains-mps

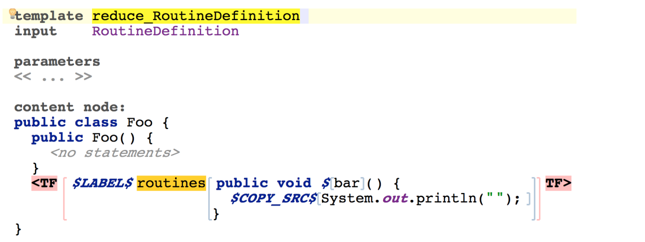
[7] http://tv.jetbrains.net/videocontent/your-second-date-with-jetbrains-mps

[8] http://en.wikipedia.org/wiki/Unification_(computer_science)

[9] http://wiki.eclipse.org/Xpand

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

[11] http://www.stringtemplate.org



```
template reduce_RoutineDefinition
input    RoutineDefinition

parameters
<< ... >>

content node:
public class Foo {
  public Foo() {
    <no statements>
  }
  <TF [ $LABEL$ routines [ public void $[bar]() {                           ]] TF>
                             $COPY_SRC$[System.out.println(""); ]
                           }
}
```

Figure 1. A template definition that replaces a `RoutineDefinition` statement with a Java method definition. The `COPY_SRC` macro replaces the **`println`** statement with the set of statements in the `RoutineDefinition`.

Java interoperability MPS has historically very strong ties to Java. MPS is itself implemented in Java, it runs on the JVM, and a Java dialect named *BaseLanguage* was the first language implemented fully in MPS; the language definition DSLs build on *BaseLanguage*. It is easy to import existing Java source code into MPS or to use Java libraries in MPS projects. MPS languages can be packaged as Java libraries and used from within Java IDEs. This all combined makes MPS feel very familiar to Java developers.

Over time MPS has collected numerous *BaseLanguage* extensions, such as collections, closures, time/date manipulation, regular expressions or builders, each enhancing its capabilities far beyond plain Java. Together with the fact that developers can build their own domain-specific extensions of *BaseLanguage*, this makes MPS an interesting solution for Java developers. At the same time, MPS' domain of applicability is not limited to Java and JVM. The mbeddr project, for example, successfully managed to expand MPS into the domain of C-based embedded software development.

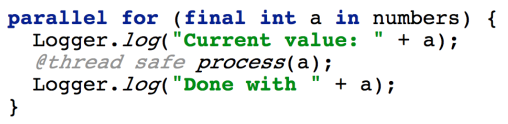
IDE interoperability The MPS workbench is primarily a tool for language definition. Languages and DSLs get designed, tested and debugged in MPS and then packaged and distributed to language users, i.e. application developers. Since typical application developers do not need all the MPS language design capabilities and prefer their existing development environments, such as IntelliJ IDEA or Eclipse, MPS languages can be packaged and distributed as Java IDE plugins (IntelliJ IDEA supported already, Eclipse is planned for the end of 2013). The upcoming 3.0 release of MPS will support full cross-navigation between Java code in the Java IDE and code managed by the MPS plugin.

For users of DSLs that are not programmers (system administrators or business experts), MPS offers the possibility to generate dedicated Java applications, whose sole purpose is to support editing and maintain code using the packaged languages. The YouTrack administrator workflow console is an example of such a dedicated lightweight IDE.

3. Language Composition

Language modularization and composition is the core idea behind LOP. MPS languages can easily refer, reuse and embed one another. Since projectional editing disambiguates constructs originating from different languages, MPS does not require the construction of a unifying parser or the definition of composite grammars. Languages can be extended with new constructs and new ways to view and edit them. See [10] for a detailed discussion of language modularization.

In order to successfully build modularized languages, all aspects of language definition must be modularizable and composable. We discuss the most important ones in the following subsections. We use a parallel for loop to illustrate the approach. Fig. 2 shows how it is used.



```
parallel for (final int a in numbers) {
  Logger.log("Current value: " + a);
  @thread safe process(a);
  Logger.log("Done with " + a);
}
```

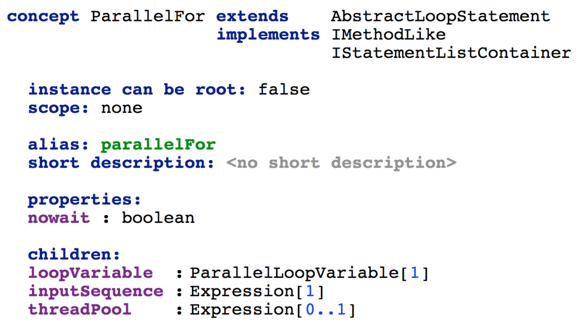
Figure 2. A sample use of `ParallelFor` in Java.

Abstract syntax Earlier in the paper we have made the association between the language concepts and objects in object orientation. This analogy also holds for language extension. Concepts can extend by other concepts, and subconcepts can be used polymorphically. Fig. 3 shows how the parallel-for loop extends the **`AbstractLoopStatement`** concept, which is defined in *BaseLanguage*. Since **`ParallelFor`** can now be used wherever **`AbstractLoopStatement`** is expected, the new extension will seamlessly integrate into the original language. Just like methods on objects in OOP, *behavior* methods attached to concepts can be overridden by their subconcepts and their constraints may be altered.

The *Adapter* pattern [5] for language composition uses this extension mechanism to allow for other types of language integration – *reuse*, *referencing* and *embedding*. For example, to embed an existing language of mathematical formulas into Java, we'd have to create an adapter (language) that wraps the formulas or parts of them into Java language concepts, such as **`Statement`** or **`Expression`**. Once wrapped, the formulas can become part of the Java abstract model, can be rendered and edited on the screen and also generated, using the reduction rules defined by the wrappers. The adapter language would also define *scoping rules* for the formulas so that they could refer to one another or to, for example, Java variables in the surrounding Java code.

Concrete syntax The MPS editor assigns visual notations to language concepts; each concept is responsible for its own rendering and user interaction. This mechanism works irrespective of the language the concepts has been defined in. So a math formula will render itself correctly, no matter whether it is part of a Java program or an electrical circuit simulation model, for example. Subconcepts inherit the editor of their parent concept, unless they define their own, more specific editor. Extending languages may supply their own editors for inherited concepts and thus override the concrete syntax derived from the inherited editor.

IDE functionality It is possible to alter many of the IDE aspects of a language through language extension. Lan-



```
concept ParallelFor extends    AbstractLoopStatement
                    implements IMethodLike
                               IStatementListContainer

  instance can be root: false
  scope: none

  alias: parallelFor
  short description: <no short description>

  properties:
  nowait : boolean

  children:
  loopVariable   : ParallelLoopVariable[1]
  inputSequence  : Expression[1]
  threadPool     : Expression[0..1]
```

Figure 3. Structure definition for `ParallelFor`.

guages may provide additional refactorings, intentions, editor actions and others.

Typesystem Together with new concepts, languages may introduce new types and relate them to types of other languages through subtyping rules. All type equations from all used languages enter the pool of rules that the type system engine resolves. This declarative approach makes the type system easily extensible.

Generator By extending the generator, extensions can alter the semantics of the original language. The MPS generator resolves the generator rules and mapping configurations from all languages attached to the project and builds a global generation plan. The plan specifies the execution order for the generator rules based on their mutual relative priorities expressed in mapping configurations. This enables language extensions to inject their own desired generation rules into the most suitable generation phase. Since priorities are expressed as a collection of relative ordering between mapping configurations, a language extension does not need to know about all other generators involved in the generation of a particular model. Potential (and rare) clashes are detected and left to the developer to resolve. Once created, the generation plan is used to iteratively invoke the generators, potentially leveraging the parallelism of the underlying hardware for mutually independent rules.

Providing additional reduction rules is one way to extend a language. Using a *generator switch* is another option. If the parent language uses a generator switch to choose the right reduction rules, the language extension may extend that generator switch with its additional logic for picking the reduction rules – typically to include new rules contributed by extension languages.

4. Related Work

Parser-based systems Projectional editing makes MPS a rare species, together with the *Intentional Domain Workbench* and *oomega*[12]. Although it is not a new idea, text editing is more widespread. To name a few workbenches from the parser camp, we should definitely mention *Xtext* and its related projects - *Xbase*, *Xpand* and *Active annotations*. *MetaEdit+* and *Spoofax* are also good examples of workbenches built around parsing and text editing.

Text-based workbenches feel more familiar to today's developers, as editing text is the mainstream approach to writing code. Also, integration with the existing development tool chain, such as VCSs, code review tools or code sharing facilities, is pretty straightforward when the sources are text. On the other hand, text-based tools restrict the syntax of languages to textual notations. Tables, math symbols or graphical representations cannot easily be included. For the same reasons, these tools typically do not let developers switch between alternative notations. Finally, defining robust composable languages is extremely difficult to achieve with grammars [1, 6]. In a similar vein, domain experts, who typically are the target audience for DSLs, and who do not necessarily have a strong background in programming, may find projectional editing more convenient, as they do not carry the baggage of old habits and since projectional editors can be easily made to resemble the tools or notations these experts use in their daily practice. Refer to [11] for more details on contrasting the two types of workbenches.

[12] http://www.oomega.net/produkt/

Internal DSLs in modern GPLs The raising popularity of alternative JVM languages, such as Kotlin, JRuby, Scala or Groovy can be understood as an indication of Java, the language, not satisfying today's developers' needs. Concise syntax for collections and maps, closures, data/time manipulations, regular expressions are a few examples of where the newcomers beat Java in expressiveness and ease of use. One important contribution of these languages is their flexibility towards creating internal DSLs [7]. Internal DSLs are a useful way to increase expressiveness of a GPL with little or modest effort. Using the language-specific run-time or compile-time mechanisms, the language can be extended to express common idioms in a more concise way. However, these DSLs are limited – to various degrees – by the host language's syntax and they cannot extend the host language type system. Finally, the IDE support for such DSLs is either ignored completely, or very limited.

5. Summary

MPS is a powerful tool for language development with special focus on easy language modularization and composition. In this paper we explored the way to extend languages in MPS and to customize language IDE support. The MPS projectional editor is an important advantage that not only offers notational freedom, but also enables flexible composition of languages. Thanks to its rich set of existing Java extensions and tight integration with Java IDEs, MPS enables easy interoperability between plain Java, *BaseLanguage* and DSLs. Despite its relatively steep learning curve, MPS could help the adoption of the LOP principles among software practitioners and move the industry forward by making programming languages truly customizable.

References

[1] M. Bravenboer and E. Visser. Parse table composition. In R. L. In D. Gasevic and E. V. Wyk, editors, *Revised Selected Papers of SLE 2008, Toulouse, France. LNCS Vol. 5452*, 2009.

[2] S. Dmitriev. LOP - Language Oriented Programming: The Next Programming Paradigm. 2004.

[3] T. Fechtner. MPS-based Domain-specific Languages for real-time Java development. 2004.

[4] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005.

[5] M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.

[6] T. Goldschmidt. Towards an incremental update approach for concrete textual syntaxes for uuid-based model repositories. In R. L. In D. Gasevic and E. V. Wyk, editors, *Revised Selected Papers of SLE 2008, Toulouse, France. LNCS Vol. 5452*, 2009.

[7] D. Gosh. *DSL in Action*. Manning Publications co., 2010.

[8] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA*, pages 451–464, 2006.

[9] V. P. Valeria Adrianova, Maxim Mazin. MPS use of YouTrack case study, 2012.

[10] M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain Specific Languages*. Markus Voelter, 2012.

[11] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *Proc. of SPLASH 2012*. ACM.

[12] M. Voelter and K. Solomatov. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *SLE 2010*, LNCS. Springer.