JetBrains MPS: Why Modern Language Workbenches Matter

Václav Pech

Abstract The goal of this chapter is to give a perspective on language workbenches, as well as to provide an overview of the MPS features. It starts with an introduction to language workbenches and motivations for DSL development. It then continues with an overview of how languages are created in MPS. Projectional editing is explained and its benefits for DSL design discussed. Other essential aspects of language definition, such as language testing and migration, are covered as well. Finally, useful pointers regarding the MPS ecosystem and the user community are provided.

1 Introduction to the Domain of Language Workbenches

MPS is an open-source language workbench by JetBrains that focuses on domain-specific languages. The name of the tool, MPS, is an acronym for Meta-programming System, which emphasizes the focus on meta-programming, i.e,. creating languages and comprehensive tooling for programming.

Domain-specific languages (DSL) is a fairly established idea of using custom-tailored languages to describe programs or parts of them, such as algorithms or configuration specifications, using notations specific to a particular domain. DSLs proponents claim that using a mixture of single-purpose DSLs in concert brings benefits such as higher abstraction level, fewer errors in code, smaller technology lock-in, and better communication between developers and business people compared to using a single general-purpose language (GPL). They also state that these benefits are likely to outweigh the additional costs of language development and maintenance for many projects.

V. Pech (✉)
JetBrains s.r.o. Kavčí Hory Office Park, Praha 4 – Nusle, Czech Republic
e-mail: vaclav.pech@jetbrains.com

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_1

DSLs have been gradually gaining popularity both in academia and in industry over the past two decades. On one hand, it was various internal DSLs such as Cypher (a query language for the Neo4J graph database), Gradle (a build tool), Spock (a test framework), and many other utility-like DSLs that aimed at simplifying life for developers. On the other hand, there were external, in other words "standalone," DSLs, which do not depend on a host programming language. VHDL, MATLAB, YACC, and SQL can be considered typical examples. As Martin Fowler states in his famous article [1], external DSLs, unlike internal ones, offer much richer and more flexible notations, but typically lack advanced tooling and integrate only loosely with the rest of the codebase of a system that they are part of.

While the available IDEs can provide assistance to developers when manipulating code written in some internal DSLs, building tools for external DSLs that would be comparable in functionality with traditional IDEs requires non-trivial additional effort. Thus the greater potential of external DSLs is hindered by the lack of (or the cost of) proper tooling. This gave rise to language workbenches.

Language workbenches were created with two main goals in mind:

1. To simplify the language-definition process and help maintain the whole language lifecycle
2. To simplify or automate the process of creating tooling for the languages

A language workbench can be described as a factory for building languages and tooling for them. The tooling aspect of language workbenches must not be underestimated, especially if the DLSs are supposed to be integrated into a larger code base written in a GPL. In addition, proper tooling, especially an intuitive editor, may help non-programmers interact with the code in a DSL. This would lower the bar for domain experts to join the development team and participate in a software project as active code authors.

The tooling aspect typically includes:

- Syntax coloring in the editor
- Code completion that gives suggestions as to what code can be entered at the current cursor position
- Navigation in code (go to definition of a reference, find/highlight usages of a definition, find an element by its name)
- Error highlighting in code on the fly as well as on demand in a report
- Static code analysis that detects erroneous or suspicious pieces of code and reports them to the developer
- Refactoring (rename an element, move an element to a different location, safe delete, extract some code into a new definition, inline a definition, etc.)
- Integrated test runner to quickly test the correctness of the code
- A debugger that interactively runs a program step-by-step and allows the developer to inspect the state of the program at various moments of its execution
- Integrated support for version control systems (VCS) that offers intuitive diff views and conflict resolution
- Integrated code generator and/or a compiler that transforms the DSL code into its runnable form

- A way to split large code base into smaller parts and manage dependencies between these parts
- Offering a notion of a language version and defining a process to migrate code to a particular version of a language, ideally automatically

1.1 A Brief History of the MPS Project

MPS was created by JetBrains. JetBrains is a software company founded in 2000, which since its beginning has focused on building tools for developers. The product portfolio contains mostly IDEs for popular programming languages and platforms, including Java, Android, C/C++, .Net, Ruby, and Python. JetBrains also makes tools for team cooperation. The company has over 1200 employees and is headquartered in Prague, Czech Republic, with offices worldwide.

The MPS project was started in 2003 as an internal experiment to try innovative concepts, such as projectional editor and code generation, on the Java platform. It has gradually evolved into a regular tool that is ready for use in the industry. The open-source license permits users to use MPS in commercial as well as open-source and academic projects without fees or any liabilities for JetBrains. JetBrains offers consultancy, training, and commercial support to customers to fund further development of the tool.

MPS is a universal language workbench, and is capable of supporting a wide range of domains. Thus far, MPS has been tried in domains as varied as

- Health and medicine [2]
- Data science [3]
- Tax legislation [4]
- Formal systems specification [5]
- Automotive [6]
- Aerospace [6]
- Robotics [6]
- Embedded software [6]

MPS does not contain any restrictions or limitations that would prevent it from addressing more domains in the future.

1.2 The Business Value of Language Workbenches

There are three main benefits of using Language Engineering for building software:

- Productivity
- Quality
- Leveraging expertise

1.2.1 Productivity

The main gains in productivity come from removing repetition from code and from raising the abstraction level closer to the problem domain. Higher levels of abstraction support the programmer's focus and reduce the abstraction gap between the domain and the language. This also applies to the maintenance phase. Reading and understanding code can be greatly improved by raising the level of abstraction.

When comparing the traditional approach using GPLs with language engineering, the productivity increase comes at the price of higher initial costs—the languages must first be designed and implemented. Luckily, language workbenches often support language evolution—languages can be developed and used at the same time. Code gets automatically migrated to the most recent version of the languages. Language development can be done iteratively.

1.2.2 Quality

Code written using DSLs tends to contain fewer defects, mainly due to the following reasons:

- The code is shorter than when written using a GPL.
- The code is easier to analyze by tools and reviewed by humans.
- The language can forbid dangerous or suspicious code constructs, e.g., pointer manipulation or null values.
- Error messages that the IDE shows to the user can be domain-specific, e.g., "Symbol is not available on a phone keyboard."
- The generator generates code with consistent quality—new features (aka business rules, menus, etc.) have the same quality as the old ones, since the generator used to generate runnable code for them is the same.

1.2.3 Leveraging Expertise

Language engineering separates the concerns of the problem domain and the implementation domain.

Problem domain—is covered by the user models written in DSLs. Domain experts understand this domain and thus benefit from being able to read or write code that targets the problem domain. The domain experts' expertise is encoded in the user models. These are typically preserved in version control systems and evolve at the pace of evolution of the business.

Implementation domain—is covered by the generator and the generator runtime frameworks. Professional programmers understand this domain and use GPLs to implement the logic. The logic of efficient implementation is encoded in the generator and evolves at the pace of evolution of the implementation technology (e.g., Java, the database, the operating system).

- The separation of the two concerns enables business to evolve each domain at its own pace and react to challenges in either of them independently.
- Domain experts who join the team can understand the business rules covered by the models from reading the user models, since the problem domain is not cluttered with alien implementation logic.
- Switching the implementation technology should ideally only mean defining a new generator.
- The user models, which hold the essential business logic, remain valid and usable even when the implementation technology changes dramatically.

2 MPS Terminology and Notations

People sometimes get confused about how code is represented in computer programs. Compilers, interpreters and IDEs need to manipulate code that the user has provided. Code is typically stored in text files and the extension of the file indicates the language used by that file. Computer programs, in order to represent the code in memory, need to read the files and process them with a special tool, called parser. Parser uses the known grammar of the language to distinguish individual tokens in the text file and to assign meaning to them. As parser reads the file it gradually builds a data structure that is known as the Abstract Syntax Tree. The structure represents the code fully and unambiguously. Abstract Syntax Trees (AST for short) are used by the IDEs to provide assistance to the user, and are used by the compilers to perform transformations that gradually convert the tree into runnable binary code.

2.1 Abstract Syntax Tree

MPS differentiates itself from many other language workbenches by avoiding the text form. The programs are always represented by an AST—on disk, in memory, and during transformations and code generation. This means that no grammar definitions or parsers are needed to define a language in MPS.

2.2 Node

Nodes form an AST. Each node has a parent node, child nodes, properties, and references to other nodes. The nodes that don't have a parent are called root nodes. These are the top-most elements of ASTs. For example, in Java the root nodes are classes, interfaces, and enums.

2.3 Concept

Nodes can be very different from one another—an if statement certainly looks different than a variable declaration. It is the concept of a node that codifies how a node will behave and what purpose it has in code. Each node stores a reference to its concept. The concept defines the class of nodes and coins the structure of nodes in that class. It specifies which children, properties, and references an instance of a node can have. Concept declarations form an inheritance hierarchy. If one concept extends another, it inherits all children, properties, and references from its super-concept.

While nodes compose ASTs, concepts define the possible "categories" of nodes.

2.4 Models vs. Meta-models

The terms models and meta-models are known from model-driven software design. While "models" represent user code, "meta-models" represent the "abstractions" available for creating those "models." This is somewhat similar to the distinction between "instances" and "classes" in object-oriented programming, where "classes" provide the framework and developers create "instances" of these "classes" in order to create functional systems.

In MPS "meta-models" are represented as languages, which hold concepts, and "models" are represented as user code, which consists of nodes in ASTs.

2.5 Language

A language in MPS is a set of concepts with some additional information. The additional information includes details on editors, completion menus, intentions, typesystem, dataflow, etc. associated with the language. A language can extend another language and thus define additional qualities for concepts defined in the extended language.

2.6 Modules

Projects in MPS consist of modules. Modules are independent reusable collections of code. There are four types of modules in MPS:

- Solution—represents a piece of user code and is equivalent to how code is structured in traditional IDEs.
- Language—represents language definition.

- Generator—represents definition of code transformations into another language.
- DevKit—groups modules (Solutions and Languages) for easy reference; does not add any new code or functionality. A module can be part of multiple DevKits.

2.7 Models

Internally, modules are structured into models. Models are a language-agnostic equivalent of Java's packages, Ruby's modules, or JavaScript folders. Models hold root nodes, which are a rough equivalent to files in traditional languages. Root nodes "are" the code. Technically, they represent the roots of trees (aka Abstract Syntax Tree) and hold other nodes organized into a tree-like hierarchy.

Additionally, MPS has the capacity to organize code inside models hierarchically into what are called virtual packages.

2.8 Generator

The generator gives the code meaning. It transforms the model into a model that uses a different language, typically on a lower level of abstraction. Generation in MPS is done in phases—the output of one generator can become the input for another generator in a pipeline. An optional model-to-text conversion phase (*TextGen*) may follow to generate code in a textual format. This staged approach helps bridge potentially big semantics gaps between the original problem domain and the technical implementation domain. It also encourages reuse of generators.

3 BaseLanguage

MPS comes with a clone of the Java language, called BaseLanguage. Since Java is a traditional parser-based GPL, it cannot be used directly in MPS as is, but had to be re-implemented using the MPS language definition mechanisms. This re-implementation is called BaseLanguage. The word *base* highlights that it is:

1. The fundamental language used throughout MPS to write language definitions logic.
2. A typical target of code generation—high-level DSL code is generated into BaseLanguage, which in turn can be represented as Java source code and then compiled with a Java compiler.

Although originally BaseLanguage copied Java in version 6, numerous extensions have been created over time to provide additional capabilities and bring in many of the later Java features.

However convenient, BaseLanguage is not the only desired target language for code generation. The generator can transform models between any languages, provided they all have their MPS-based definitions available. To date, MPS implementations of these traditional GPLs languages exist and are readily available—Java, XML, Html, CSS, C, C#, JavaScript, Text, PDF, and LaTex.

In order to generate other languages than these, the intended languages must first be defined in terms of MPS.

4 Projectional Editor

Editing code in MPS differs fundamentally from other, more traditional, tools. MPS is based on the concept of projectional editor, which is an invention from the 1970s that has been so far adopted mostly by tools outside of the programming mainstream. In essence, a projectional editor lets the developer manipulate the in-memory representation of the code directly, instead of letting them type characters that need to be parsed by the tool.

From the user perspective this feels somewhat similar to editing math formulas in popular text processors. The editor takes care of the structure of the code and the developer fills in the blanks. Since the code is never represented in a plain text form, there is no need for parsing text in order to build the in-memory representation of code. The code is always in this form—on disk as well as in memory.

The challenge for projectional editors is to hide the fact that the user is manipulating the AST. Historically, the editors were not very successful in making text editing convenient enough for the programming community to adopt it widely. Its applicability has thus been limited to only a few domains.

MPS has made an attempt to improve projectional editing and make it universally applicable to a wide range of possible notations. The key element of the MPS projectional editor is the idea of node transformations. When the user presses a key on the keyboard, it is not understood as a character that needs to be inserted into a text document, but instead it is handled as an event by the part of the AST that holds the cursor at that moment. A key event is announced to the registered listeners on that particular node of the AST and they will handle the event, typically by transforming the AST to reflect the character represented by the event (Fig. 1).

4.1 Notations

The notation directly influences the success of a language. People frequently adopt or refuse a language depending on how familiar they feel when interacting with the code. The notation should be simple and concise and provide sensible defaults. If a preferred notation exists already in the domain, it is advisable to reuse or adapt it in the new language.

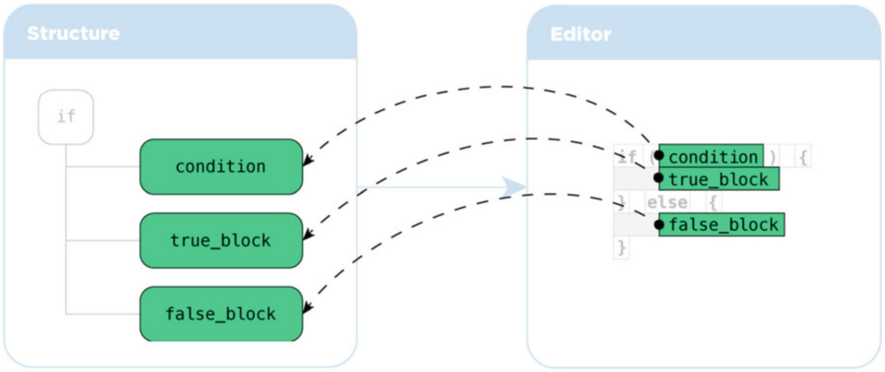


Fig. 1 Mapping of children of the concept for conditional statement to cells of the projectional editor

Finally, the support that the IDE provides (code completion, scoping, intentions, refactorings, navigation, error detection, etc.) adds to the usability of the notation itself. It is not only the static syntax of the language but also the possibility of efficient interaction with the code that users care about when it comes to language notations.

The long history of parser-based languages has made many engineers believe that textual notations are the generally preferred ones, while, in fact, they may not be optimal for numerous problem domains. Abstract Syntax Trees (AST) as a data structure are not limited to representing text. They can easily represent diagrams, tables, or anything that can be mapped to a structure of a hierarchy with references. It is the textual editor of the traditional IDEs that puts restrictions on the allowed notations. Tools such as MPS that build on the principles of projectional (sometimes called structured) editing can offer a wide range of notations.

Symbols—notations that combine text with non-textual symbols. Math, for example, has been using symbols for centuries, and there are good reasons to support these symbols in languages used in math-heavy computations.

Tables—the wide use of spreadsheets proves that tabular notations are handy for certain types of tasks, for example as spreadsheets or decision tables.

Forms—rich text editors are also frequently utilized for encoding rules of various sorts. Requirement documents, business rules specifications, and formalized contract agreements are examples of documents that can be treated as regular models with all the benefits of tooling, automation, and transformations.

Diagrams—numerous problems are best solved in graphics—electrical circuits, organizational charts, workflows, etc.

State machines—some domains, such as embedded software and robotics, use state machines to model systems that react to external events and change their internal state following a predefined set of rules. To express the transitions in individual states after the arrival of certain events a table or a diagram can be used with high success rates.

4.2 Benefits of Projectional Editing

There are three main benefits of using a projectional editor:

1. **Rich notations**—since AST can represent textual as well as tabular or graphical notations, the absence of a parser opens up the possibility of using non-parseable notations.
2. **Multiple notations**—since the persistence format of code in a projectional editor stores the AST, which is independent of the notation, it is highly possible to define multiple notations for a single language and let the user switch between them at will. The code can, for example, be presented as text when created, as a table when conducting a code review, and as a diagram when resolving a merge conflict.
3. **Language modularity**—the problem of multiple parsers is completely avoided by projectional languages. The nodes of an AST are disambiguated by the user when created and hence the tool is always certain about what concept is represented by a given node. Nodes belonging to different languages can be combined in a single code base, even in a single AST. Families of languages and their extensions can be designed in a way that permits reuse, cross-referencing, embedding, and extension of elements of one language by elements of another language. This helps to avoid the need to design bloated "all-world" language and instead encourages minimalistic purpose-oriented languages that can be combined at will.

4.3 Notations Trade-Offs

For beginners or infrequent users of a language, a chatty, highly descriptive syntax with enough visual guidance should be preferred—a learnable notation. For experienced and frequent users, an efficient, perhaps even cryptic, fast to type notation with numerous keyboard shortcuts will allow the users to achieve their tasks faster.

Orthogonally, notations can be optimized toward easy readability or writability. Ceremony in code helps in understanding context, but slows people down when they have to write the code. Compact syntaxes, on the other hand, may sometimes be challenging to read.

Since MPS enables language designers to provide multiple notations for a language, which the users can choose from when editing code, the trade-off between learnability and effectiveness, as well as between readability and writability, does not need to be made in MPS languages. Instead, languages can address these competing concerns with multiple notations, each optimized for a particular task or role. For example, when encoding a state machine, a textual notation could be used by experienced users to efficiently create definitions (weight toward effectivity and writability), while a diagram notation may be used for code reviews (readability).

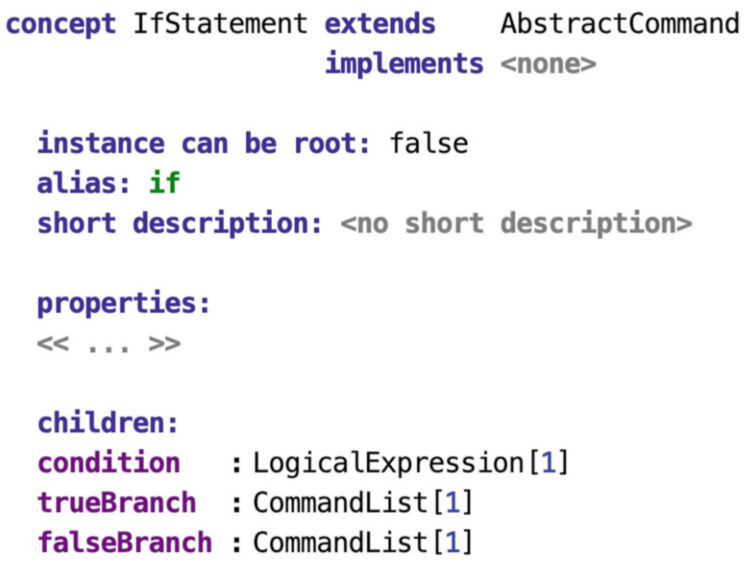
4.4 Reflective Editor

Although the editor in MPS hides the real code (AST) under the hood and instead only permits the developers to interact with the code through one of several projections, there is a way for the developer to see and touch the AST directly. MPS offers Reflective editor, which is a default visualization available for all languages automatically. It simply projects the AST onto the screen as a tree and lets the user edit the values in a default way. This is especially handy in the early prototyping stages of language development when the editors are broken or non-existent. Having such a fallback editor option might be useful even at later phases of development if none of the provided editors supports the desired editing task.

5 The MPS Way of Defining Languages

The definition of a language in MPS is split into several aspects. Each aspect is represented as an independent model inside the language definition.

- *Structure*—Holds the abstract syntax (concepts) of the language.
- *Editor*—Contains editor definitions (notations) for the concepts of the language.
- *Constraints*—Puts additional constraints on what values properties can have, what targets references can point to, and what children a node can have beyond the cardinality restriction specified in the Structure aspect.
- *Typesystem*—Holds rules that the typesystem engine should use when calculating types for nodes. Additionally, it defines checking rules that represent static code analysis checks that should be performed against models that use the language.
- *Dataflow*—Defines the control and dataflow rules for individual concepts. Based on these control and dataflows, a detailed analysis can be performed and issues such as "unreachable code" and "potential null-pointer exception" can be discovered.
- *Intentions*—Collects handy intentions that will assist the user with quick hints on how to refactor or reorganize the code.
- *TextGen*—Useful only for languages that can have their models directly converted to textual files (aka base languages). Defines rules that help MPS flush the AST into a textual buffer to create textual source files.
- *Generator*—This aspect is actually a whole module not just a model. It contains rules and templates to convert models written in this language into another language.



```
concept IfStatement extends    AbstractCommand
                    implements <none>

  instance can be root: false
  alias: if
  short description: <no short description>

  properties:
  << ... >>

  children:
  condition   : LogicalExpression[1]
  trueBranch  : CommandList[1]
  falseBranch : CommandList[1]
```

Fig. 2 The definition of a typical conditional statement in MPS

5.1 Structure

Language definition in MPS starts with the abstract syntax, which MPS calls "Structure." This is quite different from how traditional parser-based systems approach it, since these focus on the concrete syntax (aka notation) first. In MPS it is the conceptual view that is taken first—the language author defines the concepts, their properties, their references, and their mutual relationships of inheritance or aggregation (Fig. 2). The reflective editor helps a great deal with quick prototyping in this early stage of language design when editors do not exist yet or are only rudimentary.

5.2 Editor

Once the concepts are defined, their visual representation can be created. The language author is not restricted to only one visual representation of a concept; the language may offer multiple notations. One of them must be marked as "default," and the others are identified by textual identifiers (hints), allowing the final user to select these alternative notations by selecting the corresponding hint from the list.

An editor definition consists of one or more cells. A cell can contain a fixed text value, be bound to a property or a reference value of a node, or represent an editor of a child node. Cells can be styled with colors, padding, font styles, and sizes in a way similar to how CSS is used to style HTML elements. Modularization of the editor definition is allowed by using “Editor Components,” the sub-concepts of which can override and thus customize their visual representation from what they have inherited.

The ability to customize the transformations of nodes is also very important, as it directly influences the dynamic behavior of the editor. The language author can customize the content of the completion menu in given contexts, allow fluent textual editing, and offer handy visual indicators to the user.

5.3 Constraints

When a property of a concept must only allow certain values (e.g., an integer value must be between 0 and 100; a text value must meet a regular expression), it must be specified in the constraints aspect. Similarly, it should be used when the scope of a reference (the set of available targets) must be limited. Additionally, the parent-child relationship as specified in the Structure aspect may be subject to further restrictions, which the Constraints aspect allows to be specified in the “can be child,” “can be parent,” and “can be ancestor” rules. The constraints represent rules that the editor prevents from being violated. Thus the user is not allowed to insert an invalid value into a property or an out-of-scope target into a reference. Such violations are reported by the color red in the editor.

5.4 Typesystem

The typesystem serves two purposes—type calculation and static code analysis. We will briefly look into both.

5.4.1 Type Calculation

The type inference engine in MPS will attempt to assign types to all nodes in the AST. If it fails to assign a type to a node or if some rules contradict each other in terms of what type a node should have, the typesystem will report a typesystem error to the user.

The language author uses inference rules in the typesystem aspect to define rules on how types should be calculated for nodes. For example, look at the following code:

```
var a = 10
increment(a)
def increment(integer p) {return p+1}
```

Line 1 declares a variable, line 2 passes a variable reference to a method call, and finally line 3 declares the method called increment. In a straightforward typesystem implementation for such a language there would be the following inference rules involved in computing the type of the variable “a”:

1. The type of “10” is integer.
2. The type of a variable declaration equals the type of its initializer.
3. The type of a variable reference equals the type of the variable declaration that it references.
4. The type of the parameter of method “increment” is integer.

These rules are enough for the typesystem to annotate completely and unambiguously all nodes in the code snippet with types. Additionally, a check should be inserted into the ruleset that, once the types are calculated, will verify that we are passing an integer into the “increment” method: “The type of the argument when calling the increment method must equal to the type of the parameter in the declaration of the method.”

In this way the typesystem can detect situations when a value of an invalid type is being passed into the method.

5.4.2 Static Code Analysis

The typesystem offers checking rules to define code structures that should be reported to the user. This is typically used to inform the user about suboptimal code structure, potential semantics errors, unused definitions, misplaced constructs, etc. There are three levels of severity to report:

- Error
- Warning
- Informational message

Unlike for constraints, the typesystem checking rules do not prevent problems from happening. Instead, they report already present problems to the user and allow the user to trigger an associated quick-fix, if available, that will correct a particular issue.

5.5 Generator

While languages allow their users to create code, which MPS stores in models, generators can transform these source models into target models. Generators perform model-to-model transformations on models. The target models use different languages than the source models (typically the abstraction level is reduced/lowered during a generation transformation) and serve one or more purposes:

- Models can be converted to text source files and then compiled with standard compilers (Java, C, etc.).

- Models can be converted to text documents and used as such (configuration, documentation—property files, XML, PDF, html, latex).
- Models can be directly interpreted.
- Models can be used for code analysis or formal verification by a third-party tool (CBMS, state-machine reachability analysis, etc.).
- Models can be used for simulation of the real system.

It is possible for there to be several generators for the same DSL, each generating different implementation code. These generators may all be used at the same time to target multiple platforms, or they may evolve over time, one replacing the other as the needs for the run-time platform evolve. Regardless, it is sensible to follow this rule: A generator must not influence the design of the language.

The lifetime of a generator is usually shorter than the lifetime of the language and the models created with it. It is the user models that hold the biggest value for the customer as the user-written models contain the business knowledge collected over time by domain experts. This value must be preserved even if the implementation platform of the system may change over time.

A generator consists of stable and variable parts. The stable parts is the code that is always generated unchanged, no matter how the user implements the logic in the DSL. Large portions of the stable parts are frequently extracted away from the generator and implemented as a library that bundles with the generated code at run-time. The variable parts, on the other hand, reflect the actual DSL code. The generator thus has to support parametrization of the generated code to reflect the values in the input model.

The generator in MPS takes a templating approach. Each generator contains a set of rules that specify what templates to apply to what nodes in what context. The generator also contains the templates that specify code snippets in the target language into which a node should be translated.

When, for example, generating code written in language A into code written in language B, the generator author must specify rules that for each concept in language A define a template written in language B, which should be used to replace nodes of this concept in code with snippets of code in language B. These templates are parameterized with values from nodes of the original model. The parameterization is done using a family of macros that annotate the nodes of language B in the templates or their properties, children, and references, and specify how to set values depending on values in the input model.

6 Integration with Other Systems

6.1 Language Plugins

MPS builds on the modular IntelliJ platform. Its functionality can be enhanced through plugins that the users install through the UI, either from the JetBrains plugin repository or directly from a plugin zip file. This is a convenient option for languages to be shared among developers—a language or a set of related languages is packaged as a plugin for the IntelliJ platform and then distributed to the users, either directly or via the JetBrains plugin repository. On the receiver end it is just a matter of choosing the plugin from a list in the MPS UI and the languages can be instantly used to write models.

6.2 Standalone IDEs

Being a language workbench, MPS is a fairly large and complex tool. While it is acceptable for language designers to manage its complexity, for the intended users of the languages themselves a much simpler tool is typically needed. All the language design tooling and UI elements can safely be dropped in a tool, the single purpose of which is to let the user create and edit models in some languages and possibly generate them into the desired implementation code. By leveraging the IntelliJ platform, MPS enables the language authors to package a language or a collection of languages together with the core IDE functionality into a single-purpose tool—a custom IDE. This Java-based IDE, which is a single-purpose modeling tool, is then distributed to the users. The tool holds very little MPS-related heritage.

This is the favorite distribution path for most languages and their associated tooling, when the target users are non-programming domain experts.

6.3 Build Language

MPS has its own build automation facilities that allow language authors to create modular descriptions of how their languages should be built, what dependencies they have, and into which locations the output should be packaged.

The MPS build language is an essential component of this whole process. Build Language is an extensible build automation DSL for defining builds in a declarative way. Generated into Ant, it leverages the Ant execution power while keeping the sources clean and free from clutter and irrelevant details. Organized as a stack of MPS languages with Ant at the bottom, it allows each part of the build procedure to be expressed at a different abstraction level.

The build scripts can be used to create language plugins and standalone custom language IDEs, and to run language tests. Since Ant was chosen as the implementation technology for the MPS build scripts, all of these tasks can be performed from the command line as well as automatically on the continuous integration server. This enables MPS to participate in the automated build processes and thus produce or consume artifacts that together form a complete software application.

6.4 Persistence

Since all code in MPS persists in its AST form, a hierarchical structure must be present in the file that represents an MPS model on disk. By default, XML is utilized by MPS, with a proprietary binary format being an alternative. Custom persistence formats, such as database, JSON, and custom-formatted text files, can be plugged into MPS by providing a model serializer and a de-serializer.

6.5 Version Control

MPS leverages the IntelliJ platform to integrate with the most popular VCSs such as Git and Subversion. Since the users do not expect to manipulate code in its raw persistence format (XML or another structured format), MPS provides its own UI for diff view as well as for conflict resolution. The projectional editor is displayed to the user to render the particular code version. If multiple projections are available in the language, the user can conveniently choose which projection to use in the view.

6.6 Third-Party Tooling

From the software architecture point of view MPS is a modular Java application. It enables developers to enhance it with additional tools and useful visual elements. Typical examples include:

- External code analysis tools that need to be triggered from MPS in order to check the MPS models
- Code verifiers that need the models to be transformed into a particular format, then perform the verification, and finally present the results to the user in an intuitive way
- External debuggers
- Additional model visualizations that provide the user with a different view on the models
- Intuitive visual reports or consolidated statistics calculated from the models and presented to the user in an arbitrary way

7 Language Versioning and Migrations

Languages, like any piece of software, need to evolve over time. When changes are introduced into a language, existing code may break. Migrations automate the process of fixing such broken code by upgrading it to the recent version of the language.

After a language has been published and users have started using it, the language authors have to be careful with further changes to the language definition. Some changes to a language break code; some don't. In particular, removing concepts or adding and removing properties, children, and references to concepts will introduce incompatibilities between the current and the next language versions. This impacts the users of the language if they switch to the next language version. Failures to match the language definition will be presented to the user as errors reported in their models.

Breaking changes should always be accompanied by migrations to avoid problems on the user side. Migrations may also be useful for user convenience to automatically leverage non-breaking changes in their models.

Non-breaking changes:

- Add a new concept.
- Add properties to a concept.
- Add children and references to a concept, provided they are optional.
- Rename concepts, properties, children, and references.
- Loosen constraints.

Breaking changes:

- Add mandatory children and references (cardinality 1, 1...n).
- Remove concepts, properties, children, and references.
- Tighten constraints.

MPS tracks versions of languages used in projects and provides automatic migrations to upgrade the usages of a language to the most recent version of the language. The language designers can create maintenance "migration" code to run automatically against the user code and thus change the user's code so that it complies with the changes made to the language definition. This is called language migration.

8 Testing Language Definitions

Modern software development relies on automated testing as a way to increase reliability. The same holds true for languages in MPS—many different aspects of the language definition can be tested automatically. Structure, editor, constraints, typesystem, scoping, intentions, migrations, and the generator can all be tested with automated scripts. Let's cover the individual options that developers have in order to test the language definitions.

8.1 Debugging

MPS comes with a Java debugger and offers integration points for other external debuggers. Since MPS languages are implemented in Java, the built-in Java debugger can be used to debug language definitions. Extensive support is also available for trace and log messages to be inserted in code and then explored in the log.

8.2 Editor Tests

Editor tests allow language designers to test the editor definition and its reaction to user actions. Each editor test consists of three parts:

1. A starting piece of code and a starting cursor position
2. An expected resulting piece of code and optionally a resulting cursor position
3. A sequence of user-initialized events, such as text typed, keys pressed, or actions triggered, plus assertions of the editor state

Editor tests are run against the desired MPS editor definition, while the user actions are simulated and the resulting code is compared with the expected result. Additionally, the test can make assertions against the state of the editor context at any moment during its execution. This can be handy, for example, to test the available options in a completion menu or the visibility and properties of a particular editor cell.

8.3 Node Tests

Node tests focus on testing the Structure, Constraints, Typesystem, and Dataflow aspects of language definition. The test contains a piece of code together with assertions about its correctness. These arrestors can either be inserted as annotations into the code itself or expressed in an imperative style in one or more test methods. The assertions typically check:

- The presence or absence of errors on nodes
- The presence or absence of warnings on nodes
- The type calculated by the typesystem for a node
- The scope of a reference
- Violations on constraints

8.4 Migration Tests

Migrations tests can be used to check that migration scripts produce expected results when run against a specified input. A migration test can test a single migration or multiple migrations applied together. The test defines:

- The migration or multiple migrations to apply
- The nodes to apply the migrations to
- The nodes that represent the expected outcome of the test

8.5 Generator Tests

Generators can be tested with generator tests. Their goal is to ensure that a generator, or set of generators, conducts its transformations as expected. As with most tests in MPS the user specifies:

- The pre-conditions in the form of input models
- The expected output of the generator in the form of output models
- The set of generators to apply to the input models

A failure to match the generator output with the expected output is presented to the user in the test report.

9 The MPS Community

The community around MPS has been growing steadily since the early days of MPS. At first it was formed mostly by innovators who were seeking efficient ways to create complex software systems. Many of these luminaries, such as Markus Voelter [7] and Fabian Campagne [8], helped spread the word among early adopters. Soon interesting experimental projects started popping up.

- mbeddr [9]—an extensible C implementation for embedded software development
- MetaR [10]—an R IDE for people with limited computer science background
- NYoSH [11]—a tool designed as a modern replacement for Unix/Linux command line shells
- iets3 [12]—base language for system modeling and specification including basics abstractions for components, expression, variability, etc.

This pioneering work led to first customers adopting MPS for their business. Many of these describe their projects later in this book.

9.1 Sources of Information

There are several places where MPS community members go for information and advice:

- Online forum [13]—this is a discussion forum where people can ask and answer questions related to the MPS technology as well as to language design in general.
- Blog [14]—the MPS team regularly informs the community about new releases, interesting features, and upcoming events through the blog.
- Slack [15]—the official slack channel, where the users can talk to the developers as well as the other members of the MPS community.
- MPS rocks [16]—an informational portal on everything about MPS run by Kolja Dummann.
- Publications page [17]—MPS, being open-source software, encourages researchers from academia to use it for their experiments with languages. The MPS team maintains a list of relevant papers that have been published by people from academia as well as from the industry.

9.2 MPS Extensions

The mbeddr team has gradually built a comprehensive collection of utilities and utility languages that were not specific to their primary domain of focus, which was embedded systems. They generously extracted these handy gadgets that solve many recurring problems in language engineering into an independent package and shared it with the community. The library is available as a download [18] separate from MPS itself. In addition, the individual tools have been deployed as plugins into the JetBrains plugin repository for easy installation.

9.3 Language Repository

The collection of publicly available MPS-based languages is growing. Some of them are readily available in the JetBrains plugin repository [19]. The language repository page [20] is an attempt to keep a list of all available languages in one place.

10 Conclusion and Future Developments

The development of MPS continues. To stay relevant, MPS has to adapt to new technologies and engineering processes. The underlying IntelliJ platform covers most of these changes, although integrating some of them with the projectional editor imposes challenges. The MPS team continues to search for ways to overcome the limitations of the individual aspects of language definition:

- The constraints aspect now offers an experimental approach to constraint definition that allows the designer to componentize the individual rules and to customize the error messages.
- A new typesystem aspect will be introduced to considerably improve the expressiveness of the typesystem rules.
- The definitions of menu transformations in the editor aspect should be simplified, at least for the typical scenarios.
- The support for cross-model generation and incrementality of the generator must be enhanced.

Separately, JetBrains continues to develop a web-based projection editor that could eventually become a cornerstone of new generation language workbenches.

References

1. Language workbenches: the killer-app for domain specific languages? https://www.martinfowler.com/articles/languageWorkbench.html
2. Case Study: How DSLs transformed Voluntis in the worldwide leader in Digital Therapeutics algorithms. https://strumenta.com/wp-content/uploads/2020/05/Voluntis-Case-Study.pdf
3. The MetaR project case study. https://resources.jetbrains.com/storage/products/mps/docs/MPS_MetaR_Case_Study.pdf
4. The Dutch Tax Office case study. https://resources.jetbrains.com/storage/products/mps/docs/MPS_DTO_Case_Study.pdf
5. FASTEN.Safe: A model-driven engineering tool to experiment with checkable assurance cases, by C. Carlan, D. Ratiu, 39th International Conference on Computer Safety, Reliability and Security (SAFECOMP), 2020. https://drive.google.com/file/d/18O7iY1MkkECj%2D%2DujO-Zx5hNX7DazGMkL/view?usp=sharing
6. The mbeddr project case study. https://resources.jetbrains.com/storage/products/mps/docs/MPS_mbeddr_Case_Study.pdf
7. Markus Voelter homepage. http://voelter.de/
8. Fabian Campagne's laboratory homepage. http://campagnelab.org/
9. The mbeddr project homepage. http://mbeddr.com/
10. The MetaR project homepage. http://campagnelab.org/software/metar/
11. The NYoSH project homepage. http://campagnelab.org/software/nyosh/
12. The iets3 project source code repository. https://github.com/IETS3/iets3.opensource
13. The MPS online discussion forum. https://mps-support.jetbrains.com/hc/en-us/community/topics/200363779-MPS
14. The official blog of the MPS project. https://blog.jetbrains.com/mps/
15. The official slack discussion for the MPS community. https://jetbrains-mps.slack.com
16. The "MPS rocks" community website. https://mps.rocks/
17. The page listing academic publications related to the MPS project. https://www.jetbrains.com/mps/publications
18. The source code repository of the MPS extensions project. https://jetbrains.github.io/MPS-extensions/
19. The JetBrains plugin repository for MPS. https://plugins.jetbrains.com/mps
20. The "Language repository" page listing the important third-party languages and language plugins. https://confluence.jetbrains.com/display/MPS/MPS+Languages+Repository