

Language Workbench Challenge 2016: the JetBrains Meta Programming System

Eugen Schindler
Canon/Océ Technologies
eugenschindler@gmail.com

Klemens Schindler
Sioux Embedded Systems
klemensschindler@gmail.com

Federico Tomassetti
Independent
federico@tomassetti.me

Ana Maria Şuţî
Eindhoven University of Technology
farcasia@gmail.com

Abstract

This paper describes a solution for the [SPLASH 2016 Language Workbench Challenge \(LWC\)](#) workshop, using the [JetBrains Meta Programming System \(MPS\)](#) language workbench. As the main focus of the LWC is on implementation and not on writing a paper, we used the [mbeddr](#) documentation language to describe the solutions to the challenges posed in the LWC. In this way, the showcasing of a solution is co-located with its implementation, making it easier for the reader to try out the solutions and to better understand them. Therefore we would encourage you to actually open [the solution implementation](#) (see README.md for setting it up) and try it out.

JetBrains MPS has evolved into a powerful and flexible tool that can be used to address most of the language engineering challenges in the LWC. The feature-richness, and the scale of practical applicability of JetBrains MPS increases even more when using the [mbeddr](#) platform extensions and other plugins developed by the MPS community.

1. Introduction

In this paper, we discuss how the JetBrains Meta Programming System can be used to address the challenges presented in the Language Workbench Challenge 2016 (LWC'16).

The JetBrains Meta Programming System (MPS) is a language workbench based on projectional editing. While other language workbenches are based on projectional editing (e.g. the Whole Platform [3], MetaEdit+ [4], Intentional

Domain Workbench [2]), MPS is arguably one of the most feature-rich, with a fast growing user base.

By using a powerful and flexible language workbench such as MPS, language engineers can provide languages and IDEs for programmers and non-programmers that support a significant redesign of the processes of an organization. It is frequently the case that such processes involve various kinds of users with various backgrounds, experiences, and needs. Therefore a complete solution could potentially encompass different aspects of the internal processes, providing different views and a variety of tools such as simulators, debuggers, code-generators, and more. For example, in MPS we could create a text-like DSL to be used by developers. The same code could be projected using a graphical representation, to support discussion with domain experts. Testers could use a simulator to verify the correct behavior of the applications and documentation including graphs and tables could be generated for other stakeholders.

Considering the comparison of language workbench features presented by Erdweg et al. [1] we can see that MPS is among the most complete language workbenches. A language workbench with a comparable level of completeness is the *Whole Platform*. With respect to the feature model presented in [1], the only features missing in MPS are *Free-form editing* and *Live translation* (the second of which is currently being looked into by the [mbeddr](#) team).

1.1 Challenges presented in the LWC'16

While the call for solutions does not require to address all challenges, we attempted to cover as much as we could (given our limited time) in order to offer a broad overview of the capabilities and limitations of MPS.

The first challenge is on **Notation**. We demonstrate how MPS supports several notations. Notations are particularly relevant because they are the most visible aspect of languages. In many cases, language workbenches are used to

create domain specific languages. Each such language is intended for domain specific developers who already have their own preferred notations. Such notations are not necessarily textual. By adapting the tooling to the notations that are used daily by developers, the mental overhead of expressing their specifications and designs is significantly reduced, which increases their productivity and satisfaction in using the tooling.

The second challenge considers **Evolution and Reuse**. These are characteristics which are important for the maintenance of a solution in the long run. Any mature engineering approach should consider the whole life cycle of the proposed solution. The evolution is particularly important for languages because they are tools used to represent knowledge, probably the most valuable asset for many companies. By being able to evolve languages we can preserve the investment in building models using those languages. Reuse is another key element because it permits to significantly reduce the cost of developing complex solutions. For example, several projects based on MPS benefit from reusing languages provided as part of the *mbeddr platform*¹.

Finally, the third challenge is about **Editing**. This aspect is particularly relevant for projectional editors because they usually require users to part from the traditional textual editing experience. This transition requires significant training and it can be a cause of resistance. By improving the editing experience we can reduce this risk. While the usability of the MPS editors have been previously deemed positive by users [9] we believe is still an aspect which needs to be emphasized. By addressing this particular challenge we aim to prove the progresses of MPS in this area and highlight possible necessary improvements still needed.

1.2 Accessing the examples

The examples for these challenges are located in a repository on GitHub². To explore the examples, please clone the repository and follow the instructions in the README.md.

Many of the challenge items can be demonstrated using examples from the *mbeddr platform*³, so we have used them whenever appropriate.

We have written models in the mbeddr documentation language to present the examples. The mbeddr documentation language is a language that allows writing prose with sections, figures, embedded program nodes, etc. To open the documentation model for the examples of the *Notation* challenge, for instance, open the *Notation* model, as shown in Figure 1. The *Notation* chapter is divided into sections, each of which addresses one or more of the challenge items.

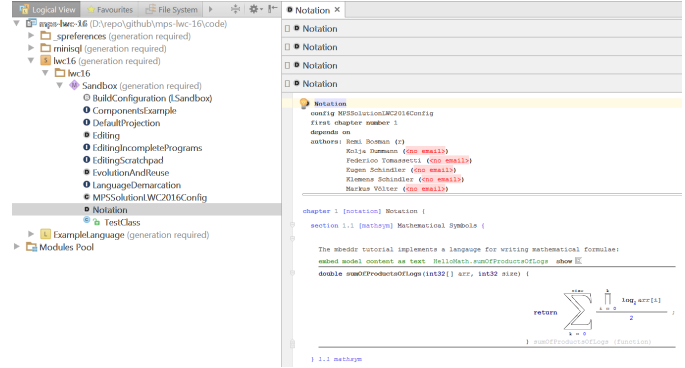


Figure 1. The documentation model for the Notation challenge

1.3 Structure of the paper

In the rest of the paper we present our proposed solutions to the three different challenges (*Notation, Evolution and Reuse, and Editing*). For each challenge we start by describing the assumptions, then we discuss the implementation, separately for each point, and finally we analyze each challenge according to the schema suggested in the challenge paper[1]: Variants, Usability, Impact, Composability, Limitations, Uses and examples, Effort (best-effort), Other comments, Artifact. Finally, we will reflect back on the three challenges in the conclusions.

2. Addressing the Notation Problem

The notation part of MPS is arguably one of its most powerful features. That is so, because almost any notation can be rendered in an editor, from textual, tabular and graphical to a mixture of these. Moreover, one can define an arbitrary UI component as a cell in the editor (the editor in MPS is made of cells⁴).

On the other hand, writing textual notations seamlessly requires more effort from the language designer because of the projectional nature of MPS. The classical example of writing a sum of two numbers in the editor by first writing the left-hand side number followed by the plus sign and the right-hand side number requires additional functions to be implemented in MPS.

2.1 Assumptions

We do not make any assumptions for the examples in the next sections. Notations and capabilities such as those described next can be provided for any MPS languages.

2.2 Implementation

In the next subsections we present the chosen examples and the implementation for each item of the *Notation* challenge.

¹ See <http://mbeddr.com/platform.html>

² <https://github.com/mps-lwc-16/mps-lwc-16>

³ See <http://mbeddr.com/platform.html>

⁴ <https://confluence.jetbrains.com/display/MPSD31/Editor>

2.2.1 Support mathematical symbols in addition to textual notation

Figure 2 shows a mathematical calculation expressed in the mbeddr math language, which is embedded in a function of the mbeddr C language.

```

section 1.1 [mathsym] Mathematical Symbols {
  The mbeddr tutorial implements a language for writing mathematical formulae:
  embed model content as text HelloMath.sumOfProductsOfLogs show ☐
  double sumOfProductsOfLogs(int32[] arr, int32 size) {
    return 
$$\sum_{k=0}^{size} \prod_{i=0}^k \frac{\log_2 arr[i]}{2}$$
;
  } sumOfProductsOfLogs (function)
} 1.1 mathsym

```

Figure 2. Mathematical notation for the mbeddr math language

This notation is possible in MPS because one can insert any drawing in an editor cell using base language code (which is a re-implementation of Java). The sum symbol, for instance, is placed in the editor component of the *SumExpression* concept, and the sum symbol is drawn on the screen using a *Graphics* object in Java. Two screenshots from the editor component of the *SumExpression* concept and the implementation of the sum symbol are shown in Figure 3.

```

<default> editor for concept SumExpression
node cell layout:
  LOOP
  lower: [ > { name } = % lower % < ]
  upper: % upper %
  body: % body %
  symbol: SumSymbolSerif
  parentheses: (node)->boolean {
    Utils.hasFollowingExpression(node.body);
  }

```

↓

```

symbol SumSymbolSerif {
  paint: (g, bounds)->void {...}

  update dimension: (dimension)->void {
    dimension.width = dimension.height * 0.7766;
  }
}

```

Figure 3. Editor component of the sum expression in the math language and the implementation of the sum symbol with some details folded

2.2.2 Support tabular notation in addition to textual notation

```

[checked]
exported statemachine FlightAnalyzer initial = beforeFlight {
  in event next(Trackpoint* tp) <no binding>
  in event reset() <no binding>
  out event crashNotification() => raiseAlarm
  readable var int16 points = 0
  state beforeFlight { ... }
  state airborne { ... }
  state landing { ... }
  state landed {
    entry { points += LANDING; }
    on reset [ ] -> beforeFlight
  } state landed
  state crashed {
    entry { send crashNotification(); }
  } state crashed
  junction NextRound {
    [points > 100] -> airborne
    [points <= 100] -> beforeFlight
  } junction NextRound
}

```

Figure 4. Textual notation for the mbeddr statemachine language with some details folded

```

[checked]
exported statemachine FlightAnalyzer initial = beforeFlight {

```

	next(Trackpoint* tp)	Events	reset()
beforeFlight	[tp->alt > 0 m] -> airborne		
airborne	[tp->alt == 0 m && tp->speed == 0 mps] -> crashed [tp->alt == 0 m && tp->speed > 0 mps] -> landing [tp->speed > 200 mps && tp->alt == 0 m] -> airborne { points += VERY_HIGH_SPEED; } [tp->speed > 100 mps && tp->speed <= 200 mps && tp->alt == 0 m] -> airborne { points += HIGH_SPEED; }		{ } -> beforeFlight
States landing	[tp->speed == 0 mps] -> landed [tp->speed > 0 mps] -> landing { points--; }		{ } -> beforeFlight
landed			{ } -> beforeFlight
crashed			
NextRound			
<>			

```

}

```

Figure 5. Tabular notation for the mbeddr statemachine language

Figure 4 shows a textual notation for the mbeddr statemachine language, while Figure 5 shows a tabular notation for the same model expressed in the same statemachine language.

There can be multiple projections associated to models of a language because MPS permits an arbitrary number of editors to be defined for the same language. For instance, the tabular notation for the *Statemachine* concept is defined in a separate editor component where the columns, rows and cells of the concept are filled in with the adequate properties, children and references of the *Statemachine* concept. An excerpt from this editor can be seen in Figure 6.

```

table editor for concept |Statemachine|
node cell layout:
[-
# exportedFlag #?|strict| statemachine { name } initial = (| %initial% |-> (| name |)) |{
[-
table {
column headers:
group "Events" {
query {
getHeaders events (node, editorContext)->join(string | EditorCell | node<> | Iterable) {...}
insert new header (node, index)->void {...}
on delete: (node, index)->void {...}
}
}
row headers:
group "States" {
query {
getHeaders states (node, editorContext)->join(string | EditorCell | node<> | Iterable) {...}
insert new header (node, index)->void {...}
on delete: (node, index)->void {
node<State> state = node.states().toList.get(index) : State;
state.delete;
}
}
}
}
cells:

```

Figure 6. Excerpt from the table editor for the *Statemachine* concept with some details folded

2.2.3 Support diagrammatic notation in addition to textual notation

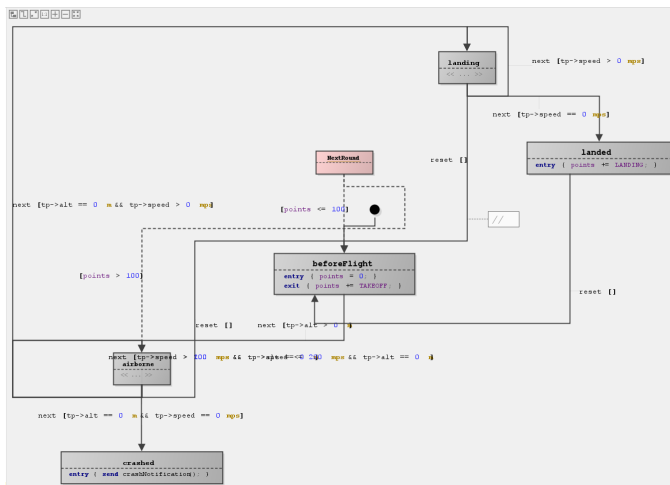


Figure 7. Diagrammatic notation for the mbeddr statemachine language

Figure 7 shows a diagrammatic notation for the mbeddr statemachine language for the same model from Section 2.2.2. The diagrammatic notation for the *Statemachine* concept is implemented in the same manner as described in the previous section, by means of an editor component where the nodes and the edges of the *Statemachine* concept are filled in with the appropriate properties, children and references of the *Statemachine* concept.

2.2.4 Support switching between multiple alternative notations for the same language

As shown in Sections 2.2.2 and 2.2.3, the notation for the same model written in the mbeddr statemachine language can be alternated between textual, tabular, and diagrammatic (the textual notation is the default notation). MPS provides support for switching between alternative notations by

means of so-called context hints. A context hint is defined for each new editor provided for a language. All the context hints defined for a language are added to a list that can be accessed from the viewer. The context hints defined for the mbeddr statemachine language are shown in Figure 8.

```

concept editor context hints statemachineStuff
hints:
  ID: table Presentation: state machine as table
  ID: graphical Presentation: state machine graphically

```

Figure 8. Context hints for the mbeddr statemachine language

2.2.5 Generic metadata annotations: annotation of program elements without changing their core meaning

Annotations are called attributes in JetBrains MPS. For defining attributes of nodes in MPS, there exists the *NodeAttribute* concept that can be attached to almost any model node by default. Thus, when declaring generic metadata annotations, one needs to extend the *NodeAttribute* concept and define its contents. An example can be seen in Figure 9 where a *GenericNote* concept is defined. To use the annotation, the language needs to be imported in the solution, and then, the *GenericNote* concept can be attached to any model node. For example, we have attached a generic note to both a paragraph in the documentation model and a Java method, as can be seen in Figure 10.

```

@attribute info
multiple: <inherited>
role: documentedNote
attributed concepts: BaseConcept
concept GenericNote extends NodeAttribute
implements <none>

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

properties:
note : string

children:
<< ... >>

references:
<< ... >>

```

Figure 9. Structure of the generic note concept

Note: This is a note attached to this paragraph
 Example annotated paragraph

Below the exact same annotation type can be seen in java code:

```

embed model content as text TestClass show
public class TestClass {
  /*package*/ int a;

  Note: Example Note
  /*package*/ int doubleNumber(int aNumber) {
    return aNumber * aNumber;
  }
}

```

Figure 10. Two example uses of the generic note in the documentation model from GitHub

The metadata annotations do not change the model elements to which they are attached at all.

2.2.6 Optional hiding: hide parts of the code, without losing the content and while retaining editability

The mbeddr environment supports product line variability [11]. In the variability language, users can define feature models (see Figure 11) and different configurations of such features (see Figure 12). In the implementation code that makes use of the features of a feature model, presence conditions guard code fragments (see Figure 13). A presence condition is a condition on features from the feature model. Once a configuration is chosen, the code that is guarded with presence conditions that do not hold is taken out by the generator. For more information on the feature models in mbeddr see Voelter et. al. [8].

```

feature model FlightProcessor
root ? {
  nullify
  normalizeSpeed xor {
    maxCustom [int16/mps/ maxSpeed]
    max100
  }
}

```

Figure 11. The feature model for flight processor

```

configuration model cfgDoNothing configures FlightProcessor
FlightProcessor_root {
}

configuration model cfgNullifyOnly configures FlightProcessor
FlightProcessor_root {
  nullify
}

configuration model cfgNullifyMaxAt200 configures FlightProcessor
FlightProcessor_root {
  nullify
  normalizeSpeed {
    maxCustom [maxSpeed = 200 mps]
  }
}

```

Figure 12. Three configuration models given the feature model of the flight processor in Figure 11

```

Trackpoint* process_trackpoint(Trackpoint* t) {
  ? [nullify]
  t.alt = 0 m;
  ? [max100]
  t.speed = 100 mps;
  ? [maxCustom]
  t.speed = maxCustom.maxSpeed;

  return t;
} process_trackpoint (function)

```

Figure 13. Function initializing a trackpoint by making use of presence conditions for elements from the feature model in Figure 11

The hiding of parts of the code is handled in the editor with the help of the *show if* option. Cells of the editor are conditioned by the *show if* option. For instance, once a variant configuration has been chosen, the editor can be updated to show the model with the processed presence conditions. Figure 14 illustrates how the viewing of the presence conditions are conditioned by a mode that is different from the compact mode. Moreover, the changes to the viewer once the mode is changed via a menu action can be seen in Figure 15.



Figure 14. The editor for concept *PreceanceCondition*

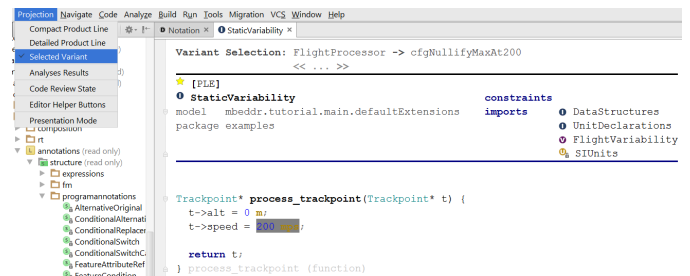


Figure 15. The *process_trackpoint* function after changing the editor mode to the selected variant; compare this with Figure 13

2.2.7 Computed properties: read only annotations that are automatically derived from the main program

The assert statements in mbeddr have an id that is read-only and that is derived from the main program. The number of the id depends on the location of the assert statement in the main program. It is used mainly in logs to easily identify failed assert statements. An example of assert statements and their ids is given in Figure 16.

```
exported testcase testFlightAnalyzer {
  FlightAnalyzer f;
  [f.init] #m_test_1a
  [assert(0) f.isInState(beforeFlight);] #m_test_1b
  [assert(1) f.points == 0;] #m_test_1c
  [f.trigger(next|makeTP(0, 20));] #m_test_2a
  [assert(2) f.isInState(beforeFlight) && f.points == 0;] #m_test_2b
  [f.trigger(next|makeTP(100, 100));] #m_test_3a
  [assert(3) f.isInState(airborne) && f.points == 100;] #m_test_3b
  test statemachine f {
    next(makeTP(200, 100)) → airborne
    next(makeTP(300, 150)) → airborne
    next(makeTP(0, 90)) → landing
    next(makeTP(0, 0)) → landed
  }
  [assert-equals(4) f.points == 200;] #m_test_4b
} testFlightAnalyzer(test case)
```

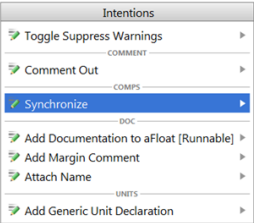
Figure 16. Assert statements and their computed ids


2.2.8 Computed structures: structured, editable views

The signatures of mbeddr component operations are an example of computed structures. Figure 17 depicts component B, which provides a port a that refers to interface A. In this component, we create a runnable (component method) with an initial dummy name, aFloat. We want this runnable to be triggered by operation a.f from the port a. When you execute the Synchronize intention on aFloat, it will get the same arguments as a.f automatically and its name will be updated consistently to portName.methodName.

```
exported cs interface A {
  float f( int32 a, float b, boolean c)
}

exported component B extends nothing {
  provides A a
  float aFloat() <= op a.f {
  } runnable aFloat
} component B
```





```
exported cs interface A {
  float f( int32 a, float b, boolean c)
}

exported component B extends nothing {
  provides A a
  float a_f(int32 a, float b, boolean c) <= op a.f {
  } runnable a_f
} component B
```

Figure 17. The synchronize intention automatically updates the runnable name and arguments.

The implementation of the synchronize intention is straightforward. Figure 18 depicts the implementation of the in-

tention. The intention is defined for *RunnableTrigger* concepts and it calls method *synchronizeRunnableParent* from concept *RunnableTrigger*. The *OperationTrigger* is a sub-concept of *RunnableTrigger* and it overrides method *synchronizeRunnableParent*. The implementation of the method is straightforward again, simply renaming the method name and adding the right arguments.

```
intention synchronizeRunnableWithTrigger for concept RunnableTrigger {
  error intention : false
  available in child nodes : false

  description(node, editorContext)->string {
    "Synchronize";
  }

  <isApplicable = true>

  execute(node, editorContext)->void {
    node.synchronizeParentRunnable();
  }
}
```

Figure 18. The implementation of the synchronize intention.

2.2.9 Skeleton editing: guide the user with syntactic templates with editable holes

The mbeddr build language is an example of a bigger structure with skeletons that can be edited. The user makes a build configuration and some course-grained choices for the build (*gcc*, *microcontroller*, *custom*, etc.), and then she has a skeleton in which the details can be filled in. An example of a build configuration is shown in Figure 19.

```
Platform
GNU paths are not checked
make: make
gdb: gdb
compiler
  path to executable: gcc
  compiler options: -std=c99
  linker options: <no link options>
  debug options: -g

Configuration Items
reporting | printf
components | generation strategy: no middleware {
  wire statically: false
  check contracts (runtime & static): true
}
state machines | trigger as const true
  generate test code false
variability mappings | fm FlightProcessor -> cfgNullifyMaxAt200
variability @ runtime
tracing
units
math
unittest | test isolation

Binaries
executable MbeddrTutorialDefaultExt is test {
  modules:
  Main
  FunctionPointers (examples)
}
```

Figure 19. A build configuration example.

Obtaining such a skeleton is possible through the editor aspect of a language. In this particular case, in the editor for the build configuration language (see Figure 20), there are three sections specified: the platform, the configuration items and the binaries. The headers of these sections are constant cells in the editor, and their constituent elements can be filled in by the user once the build configuration is instantiated.

```
[/
# iconAndNameCell #
$ custom cell $
? NEVER GENERATE THIS MODEL
<constant>
Platform
$ custom cell $
% target %
<constant>
Configuration Items
$ custom cell $
(/ % configurationItems % /)
/empty cell: <constant>
<constant>
Binaries
$ custom cell $
(- % binaries % /empty cell: <constant> -)
/]
```

Figure 20. The editor for build configurations.

2.2.10 Embedding code in prose: mix structured code with free text

One of the most relevant examples of code embedding in prose is the documentation language in mbeddr. An example can be seen in Figure 21. The figure shows an excerpt from the file that was used to document and implement the language workbench challenge’s requirements of this year.

```
section 1.3 [codeinprose] Embedding Code in Prose (
The example below makes it possible to embed entire prose (including references)in comments of an mbeddr C
listing. As for code in prose, the best example is this presentation: the entire document is written in the mbeddr
doc language. In this document, pieces of code are embedded to make a presentation of the MPS LMC 2016 case
possible.
embed model content as text EditingUsability show @
@ EditingUsability constraints
model mbeddr:tutorial.main.plainC imports nothing
int8 aRefTarget = 0;
// A documentation comment with references @arg(data) and @arg(dataLen)
to
void SelectingAndModifyingCode(int8[] data, int8 dataLen)
{
// requires structure-aware copy/paste: copy/paste the local var into global context
int8 aLocalVariable = 10;
// does not support free-floating comments: this one plus test case doc
// requires dedicated support for commenting code: Cmd-Alt-C on the var decl
int8 aVariableThatIsReferenced = 0;
aVariableThatIsReferenced++;
// does not support custom layout: if statement; heartbleed
} SelectingAndModifyingCode (function)
```

Figure 21. Mixing prose and code.

The documentation language has as a top-level concept, the *Document* concept. The document is formed of chapters. In turn, a chapter is formed of sections and sections are formed of paragraphs. There are multiple types of paragraphs allowed in the documentation language. One of them is *ModelContentAsTextParagraph*. This type of paragraph extends *AbstractModelContentParagraph*, which contains model code pointers. Furthermore, a model code pointer contains a collection of elements that can reference any named concepts in MPS (a named element implements interface *INamedConcept*). The referenced nodes are then embedded in the editor component. Thus, *ModelContentAsTextParagraph* can embed any piece of code that has a name property. That allows us to embed an *ImplementationModule*, for instance, as can be seen in Figure 21.

Moreover, the text itself can contain actual references to concepts that are defined in mbeddr files. This is achieved with the help of the *mpps-multiline* language, the *mpps-richtext* language and the *IWord* interface [6]. If one has a concept that needs to be referenced from text (from comments or in the documentation language, for instance), then that concept needs to implement the *IWord* interface that simply presupposes to provide a string representation for the concept. An example can be seen in Figure 21, where the arguments of function *SelectingAndModifyingCode*, *data* and *dataLen*, are referenced from the comment attached to the function.

Thus, in mbeddr, one can embed both code in prose and prose in code, giving rise to a mix of the two.

2.2.11 Embedding blackboxes: allow program elements to be opaque non-textual elements

We highlight the feature on embedding blackboxes in MPS with the embedding of images in the documentation language. An example of an image introduced in a document in mbeddr is shown in Figure 22.

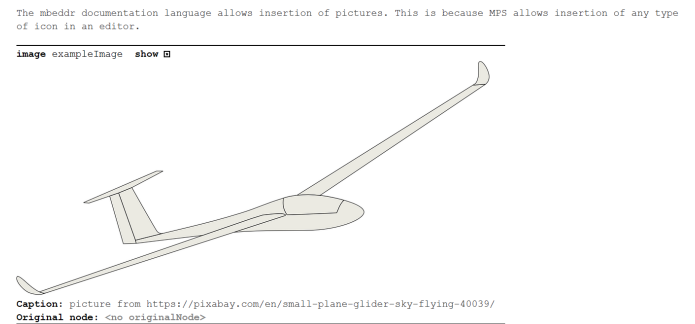


Figure 22. Embedding of an image into an mbeddr documentation file.

As discussed in the previous section, the *Document* concept is ultimately formed of paragraphs. In the mbeddr documentation language, one type of paragraph is the *ImageParagraph*. The editor component of *ImageParagraph* embeds a

swing component, and this specifies how to render an image with a given path. Part of the implementation can be seen in Figure 23. Now, once a valid image path is specified in the document, the respective image is rendered in the editor. The image itself is a blackbox from the point of view of the document.

```
<default> editor for concept ImageParagraph
node cell layout:
/|
| $ custom cell $
| $ custom cell $
|?|/
| [- image { name } ----> show checkbox { showImage } -]
| $swing components$
|/1
|
|_ ImageParagraph | ImageParagraph_presentationMode_Editor | ImageParagraph_Editor | Constraints | ImageParagraph_Behavior | check_ImagePara
|_ ImageParagraph_Editor
|_ ains.mps.lang.editor.structure.CellModel_IComponent
component provider: (node, editorContext)->JComponent {
    if (!node.resource.isValidFile()) {
        JLabel l = new JLabel("Invalid Path");
        return l;
    } else {
```

Figure 23. Part of the implementation of a swing component in the editor of concept *ImageParagraph*.

2.3 Variants

There were other alternatives for showcasing some of the features presented in the previous sections. We will give here a few variants only.

Switching between notations can be accomplished not only through context hints, but with different user actions as well. In the state machine language, one can switch between different projections by using the *Projection* tab and selecting the desired projection. Details on this can be found in our documentation model on GitHub, in the *Notation* section. In the implementation in mbeddr, there is a plugin that introduces this tab and that triggers the projection selected.

There are also many other examples for the optional hiding feature. Some characteristics of concepts can be hidden and toggled with an intention, for instance. Take the export statement for modules as an example, which can be tried on modules in the documentation model on GitHub. The intention toggles the value of the Boolean property *exported*, which results in showing/hiding the export statement depending on the condition expressed in the editor of the concept (similar to the discussion in Section 2.2.6).

One other variant that can be used as an example to showcase the power of MPS is the notation in the form of a PlantUML⁵ diagram of the state machine languages. MPS allows to incorporate viewers in the editor that project a model in a language written in MPS.

2.4 Usability

The effort to make the notations as usable as possible goes to the language designer. The drawbacks of the projectional editors are well-known, but MPS goes a long way to provide tools to the language designer that help in creating a user-friendly editor [10]. The notations provided in mbeddr are a good example of user-friendly notations.

⁵<http://plantuml.com/>

2.5 Impact

Introducing notations to a language does not come with a big negative impact on the language. The notation is expressed in an editor aspect, that is provided for each concept, and this aspect does not usually influence the other aspects of the language. At most, the editor aspect might require the import of certain libraries, and that has an impact on the overall language because the import is going to be added to the dependency list of the language itself.

There are situations when the impact of the notation can be considerable on the language. If one uses the default graphical notation provided by MPS (instead of the graphical notation provided in mbeddr), one might need to change the structure of the language itself. This has a very big impact, because all the other aspects of the language depend on the structure. The changes are required if the elements that need to be represented as edges in the graphical representation are not concepts in the language itself. That would mean transforming those elements into concepts first.

2.6 Composability

Each concept in a language can have any number of editors attached to it. The different editors of a concept *C* do not interact among each others, unless one editor of *C* reuses another editor of *C* (reuse of editors for the same concept is possible in MPS). One common pattern is to define an editor component for an interface concept and reuse that editor component in the editor of a concept that implements the interface concept. This can be noticed in the mbeddr code with concept *StateMachine* and interface concept *IModuleContent*, for instance.

Composability is very well supported at the editor level, because each editor is made of cells, and cells, in turn, are formed of other cells. This permits defining the editor of a concept as a collection of cells, where some cells are the editors of the children of the concept.

Moreover, a sub-concept inherits the editor (the notations) of its super-concept. If this is not the desired behavior, the editors of the sub-concepts can be overridden as well.

2.7 Limitations

The notations introduced can be used only inside MPS. That applies also to the textual notations, because the files are persisted as XML on disk, and they are projected as text on the screen.

Moreover, as already mentioned, there are certain limitations inherent to projectional editors especially when it comes to textual notations, but this limitations are well mitigated by MPS [10].

2.8 Uses and examples

The uses and examples have been covered in the implementation section. Almost all examples in that section are from mbeddr, with the exception of the generic metadata annotation, where we created a separate language with the generic

note concept. This language and the generic note concept can be found on the GitHub repository of our submission.

2.9 Effort (best-effort)

Some of the items described in the previous sections are trivial to address (in the order of a few minutes):

- The skeleton editing is provided by default in JetBrains MPS and is a natural consequence of how editors are created in JetBrains MPS.
- The switching between notations is a matter of introducing one context hint; that is, one file and a name for the context hint.
- The optional hiding is a matter of writing one condition in the editor cell that should be hidden.
- Embedding code in prose is trivial if we want to make references to elements defined in code, because it requires implementing one interface and defining one operation.
- Computed structures and properties such as the one exemplified imply the implementation of a function that computes the necessary structures.
- Generic metadata annotations are trivial to add to model elements as well, because they involve only defining a new concept that extends the node attribute.

Other items require more time (in the order of a few minutes to a few hours, depending on the size of the language):

- The tabular notation can be more involved because it requires that a few functions are provided to fill in the columns and the rows of the table.
- The same holds for the diagrammatic notation where code needs to be filled in that specifies what concepts are going to fulfill the role of edges and nodes and under what conditions.
- Embedding entire chunks of code in prose requires more effort than simply referencing an element from code in text.

More involved items that require considerable time (in the order of a few hours to a day):

- Embedding blackboxes, such as images, requires that one implements the logic to render the image in the editor and the logic to read the path of such images.
- Showing math symbols in the editor takes more time as well, as the rendering of the symbol in the viewer needs to be implemented.

2.10 Artifact

The examples discussed in this section can all be found in the documentation model in the *Notation* section. As for their implementation, that resides in the mbeddr code (see file *README.md* for the version of mbeddr we used).

3. Addressing the Evolution and Reuse Problem

In this section we present our solution to the challenge on Evolution and Reuse.

3.1 Assumptions

Our solutions are presented making references to the two main language families developed for MPS.

First of all we have the *BaseLanguage*, which is an implementation of Java built from JetBrains and shipped with MPS. Several extensions are distributed with MPS, including extensions to support lambdas, manipulation of sequences, expressions to access MPS concepts and models, etc.

Another very popular family of languages is *mbeddr*. It is an implementation of the C language in MPS with support for variability, state machines, testing, documentation and more. In addition to that, the *mbeddr* platform has been created. It is a collection of language extensions not specifically related to *mbeddr* or the C language. These extensions proved to be useful in different contexts.

3.2 Implementation

In this subsection we present how we implemented a solution to each specific challenge.

3.2.1 Language extension: modularly extend a language with new syntactic constructs

In this section, we are going to describe the *mbeddr* language for state machines with event-driven execution. The *mbeddr* state machines extends the base language from MPS. This enables a seamless integration between C code and state machine specifications.

State machines are a mathematical model of computation often used in embedded software for describing discrete behavior through state transitions. Its characterizing ingredients are states, transitions and events. At any given time, a state machine is in a state and it can be transitioned from one state to another. A transition in a state machine is triggered by an event. These events are usually provided by the environment, and, hence, the state machine needs to have a way to interact with the environment. Besides events, transitions can also have different guard expressions that need to hold when the event arrives, for the transitions to be triggered.

We are now going to describe the extension of *mbeddr* with new syntactic notations for state machines.

In *mbeddr*, the state machine language is packaged in a language module and it extends the base language of MPS. The *StateMachine* concept extends *BaseConcept*, which means that the state machine is a program node, as *BaseConcept* is the concept from which all other concepts are derived. The state machine language also implements the *IModuleContent* interface, which means that they can be top-level components in modules or can be inside of any container

that expects *ModuleContent* children. Modules in mbeddr C introduce basic program modularization, visibility control and namespaces [8].

In the next paragraphs, we are going to present excerpts from a state machine for judging flights. The state machine awards points for successful takeoff and landing and for speed flown [7].

The state machine adds custom notation for specifying the state machine. The textual form of the state machine can be seen in Figure 24. The figure depicts a hierarchical state machine that computes the points for a flight. In addition, because textual forms can live alongside graphical and tabular forms in MPS, the state machine can be viewed in table form and graphical form as well alongside the piece of C code where it is used. Figure 25 shows the same state machine for flight analyzes in tabular form.

```
exported statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  macro stopped(next) = tp.speed == 0 mps
  macro onTheGround(next) = tp.alt == 0 m
  in event next(Trackpoint* tp) <no binding>
  in event reset() <no binding>
  out event crashNotification() => raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    entry { points = 0; }
    on next [tp.alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  } state beforeFlight
  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight { points = 0; }
    on next [onTheGround && stopped] -> crashed
    state flying (airborne.flying) {
      on next [onTheGround && tp.speed > 0 mps] -> landing
      on next [tp.speed > 200 mps] -> flying { points += VERY_HIGH_SPEED; }
      on next [tp.speed > 100 mps] -> flying { points += HIGH_SPEED; }
    } state flying
    state landing (airborne.landing) {
      on next [stopped] -> landed
      on next [ ] -> landing { points--; }
    } state landing
    state landed (airborne.landed) {
      entry { points += LANDING; }
    } state landed
  } state airborne
  state crashed {
    entry { send crashNotification(); }
  } state crashed
}
```

Figure 24. Hierarchical flight analyzer state machine - textual notation

Moreover, the state machine itself embeds arbitrary code in the actions and in the guards. The actions are statement lists and the guards are expressions. For instance, look at the guards and actions in Figure 24; they contain mbeddr C expressions.

3.2.2 Language embedding: embed a separate language inside another

We have implemented a simple toy language representing a subset of SQL. This language permits to define database schemas and simple SQL statements referring to such schemas.

Typically SQL is used in combination with General Purpose Languages (GPLs): from the GPLs, queries are gen-

erated by filling out SQL templates with variable elements. The results of these queries are then possibly processed using GPL code.

In our example we implemented both embedding of C code into our MiniSQL and embedding of MiniSQL into C. By embedding C code in MiniSQL we can define SQL statements with variable elements. For example we can refer to a C variable containing an ID in our SQL statement. In this way we can vary the value of the variable to obtain parametric SQL queries. We can then execute those queries using libraries such as those based on ODBC⁶. It is important to notice that the MiniSQL embedded in C can still be edited with proper support regarding validation and auto-completion. It is not “just a string”. This technique is illustrated in 26.

```
void* getAuthorById(int16 id) {
  string s = sqlCode: SELECT *
                        FROM Authors
                        WHERE author_id = c`id`
  return runQuery(s);
} getAuthorById (function)
```

Figure 26. Embedding SQL code into C code and vice versa

To demonstrate the flexibility of this approach we have also embedded our MiniSQL into the BaseLanguage, which is basically an implementation of Java in MPS. You can see it in 27. Two separate extensions permit to embed the same language into different hosts (C and Java). For further explanations on this approach refer to [5].

```
public Object authorById(int id) {
  query(sqlCode: SELECT *
              FROM Authors
              WHERE author_id = j`id`
  );
}
```

Figure 27. Embedding SQL code into Java code and vice versa

3.2.3 Extension composition: combine independently developed extensions

The mbeddr project contains many examples of language composition. Different extensions have been developed during the years, not necessarily from the exact same persons or in the context of the same project. However all of these extensions can be combined and used together.

A very interesting example is the Documentation language which permits to add references to other portions of code or to embed specific constructs into the documentation.

⁶See https://en.wikipedia.org/wiki/Open_Database_Connectivity

```
exported statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
```

		Events	
		next(Trackpoint* tp)	reset()
States	beforeFlight	[tp.alt > 0 m] -> airborne	
	composite state airborne initial = flying { [] -> beforeFlight { points = 0; } [onTheGround && stopped] -> crashed flying landing landed } state airborne crashed	[onTheGround && stopped] -> crashed	[] -> beforeFlight { points = 0; }

```
}
```

Figure 25. Hierarchical flight analyzer state machine - tabular notation

In the example in Figure 28 we can see a Documentation construct containing a table. The tables extensions have been developed separately, in a completely agnostic way, so that it could be reused in very different contexts. In this case it contains expressions from mbeddr language (C). Thus, three different extensions are combined in one single example.

4.1 | Price Depends on Country and Price Group
priceDep /functional: status=accepted, @pricing

The price of the phone call depends on a number of factors. Among them are the #country and the #pricegroup .

The actual #actMinPrice is computed from the #baseMinPrice with the following equation; the #priceFactor is determined by the table below: #actMinPrice = baseMinPrice * priceFactor / 100 .

		Countries			
		Germany	Italy	Spain	GreatBritain
Price-groups	PLATINUM	10	8	7	11
	GOLD	11	10	9	10
	SILVER	12	8	8	8

Figure 28. Combining tables, C expressions and the documentation language

3.2.4 Beyond grammar restrictions: disallow constructs in certain scopes, without modeling this in the (abstract) syntax

MPS offers several mechanisms to limit where a certain construct can be used. In this respect it is not limited to the abstract syntax definition, but further logic can be added to additionally constraint the concepts usable in a given scope.

For example, every *AssertStatement* is technically an mbeddr *Statement*, from the point of the abstract syntax. However, an additional rule has been defined to restricted all constructs marked as *IRestrictToTests* to be used exclusively in tests or tests related helper functions. This rule is visible in Figure 29.

```
concepts constraints IRestrictToTests {
  can be child
  (childConcept, node, link, parentNode, operationContext)->boolean {
    parentNode.ancestor<concept = ITestContext, +>.isNotNull ||
    parentNode.ancestor<concept = IFunctionLike>.testHelperFunction.isNotNull;
  }
}
```

Figure 29. Rule which constraints the usage of certain constructs to tests

3.2.5 Syntax migration: support migrating programs when concrete syntax changes

Concrete syntax changes are supported by default with projectional editing without requiring any migration. Since only the AST is stored in the model, all concrete syntax elements purely exist in the editor. Therefore, updating an MPS editor, immediately changes the presentation (concrete syntax) without requiring a change in the stored model (AST).

3.2.6 Structure migration: support migrating programs when abstract syntax changes

Unfortunately, we didn't have enough time to work out a brief example inline of this document.

Migrations in structure are handled in MPS by means of migration scripts. Whenever a metamodel for a language in the field needs to change, you can write a migration script to migrate models in order to comply with the updated metamodel. MPS automatically handles versioning of your language and detects when a model needs to be migrated.

com.mbeddr.core.unittest contains an example of a non-trivial migration script. You can explore it by opening Logical View → Modules Pool → languages → com → mbeddr → core → unittest → migrations and explore the migration scripts there.

3.3 Variants

The three first points we have seen (3.2.1, 3.2.2, and 3.2.3) have been solved through simple extension techniques. We do not see any obvious alternative for such cases. About combining independently defined extensions (3.2.3) we can consider all the projects using the mbeddr platform as examples of this technique.

Regarding the grammar restrictions (3.2.4), we implemented it specifying that certain rules can have as ancestors only certain nodes. Conversely we could have specified that certain rules could not contain specific descendants instead.

3.4 Usability

From the point of view of usability 3.2.1, 3.2.2, and 3.2.3 do not pose any issue. We simply used the mechanism of language extensions to add additional constructs to existing

language. In one case we had done that for the specific goal to embed another language, while in the other cases we did not. In all cases the new constructs can be used exactly as the existing ones, so the new constructs are as usable as the previous ones with no different interactions required from the users.

The grammar restriction presented in 3.2.4 does not pose any usability issue either. The whole mechanism is transparent to the user: elements which cannot be used in a certain context are not offered by the auto-completion mechanism and there is no immediate way to use them when they are not supposed to be used.

Migrations are typically performed through wizard dialogs. Those migrations are proposed to the user when MPS is started or the user can trigger them manually.

3.5 Impact

Language composition does not require to change existing elements.

3.6 Composability

Language composition is well supported in MPS. It is very natural and it does not require any particular technique. The only possible issues could be caused by semantic conflicts: suppose an expression language defines only statically evaluable expressions such as literals and basic mathematical expressions. Furthermore, suppose a first extension is based on this consideration and adds the possibility to display the result of such expressions. Now, if a second extension would introduce non-statically evaluable expressions, the two extensions would not interact well together. This problem could be avoided by planning for extensibility in the original language: for example we could have required each *Expression* concept to declare if it was statically evaluable or not. All the expressions of the original language would have declared themselves to be statically evaluable, while the Expressions from the second extension would have not. The extensions calculating result values could have used the method to verify that all Expressions for which it wanted to calculate a value were indeed statically evaluable and trigger an error when not-statically evaluable expressions were used in the wrong context. In this way the two extensions would interact nicely without being aware of each other.

3.7 Limitations

No particular limitations come to mind.

The only limitation we see is with migrations, because they are not reversible. This is an issue when different members of a team want to use different versions of MPS because each version comes with specific versions of the BaseLanguage: when a project is open with a new version, migrations have to be performed and these migrations make the project incompatible with previous versions. Effectively this forces everyone to use the same version of the language workbench.

3.8 Uses and examples

Language extensibility and composition are used extensively in the two well-known MPS language families: the BaseLanguage and mbeddr. The BaseLanguage defines a core language and a set of independent extensions, and mbeddr does the same.

3.9 Artifact

The implementation can be found on a dedicated github repository⁷.

4. Addressing the Editing Problem

4.1 Implementation

In the next subsections we present the chosen examples and the implementation for each item of the Editing challenge.

4.1.1 Editing incomplete programs: support for syntactically malformed programs

```
void <no name> () {  
    if (<condition>) {  
  
    } if  
} null (function)
```

Figure 30. An incomplete function with missing name, containing an if statement missing a guard.

It is possible to edit and persist an incomplete model, however there are some restrictions to this. A node, for example an if statement, must have a complete skeleton. It is possible to leave content out such as the guard and body in an if statement. The result can be considered syntactically incorrect since the guard is missing from the if statement. However, the construction is still structurally sound since it is a valid tree node, albeit with some gaps to be filled in. Figure 30 shows an example of a function with omitted name containing an if statement missing the guard.

```
void cannotDeleteOnlyClosingBracket() {  
  
} cannotDeleteOnlyClosingBracket (function)
```

Figure 31. Since the closing bracket is not part of the model, but the presentation, the bracket cannot be removed.

Removing arbitrary pieces of "text" from the model is not possible. For example the closing bracket of an if statement can not be removed in isolation. The reason for this is that it is not part of the model contents (AST), but only of the presentation (concrete syntax).

⁷ <https://github.com/mps-lwc-16/mps-lwc-16>

```
void expressionMissingAParenthesis() {
    int16 a = 5 + (4 - 4 / 3;
} expressionMissingAParenthesis (function)
```

Figure 32. An unmatched parenthesis element enables more text-like editing. Inserting a closing parenthesis restructures the AST.

Designing language and editors in MPS can enable a text-like editing experience. For example, it is possible to design elements such as an "Unmatched Parenthesis" into your language which are not intended for the final model, but serve as an aid to enable a text-like editing experience. Figure 32 shows an example of such a scenario.

4.1.2 Structure agnostic copy-paste: copy-paste works across syntax boundaries

MPS allows selecting, copying, and pasting nodes across any language boundaries, however selection of nodes must follow the tree structure. For example, it is possible to copy 5 full statements from a selection. It is also possible to copy a node which contains a component (component language) containing a state machine (statemachine language) containing c code (c language) in action implementations. However, it is not possible to start selection halfway a node and end it halfway another one.

It is also possible to copy parts of a text-like language into a text editor. The textual content will be in the text document, but layout information and whitespace may vary.

4.1.3 Restructuring: changing syntactic structure without typing the complete expression again.

The example in Figure 32 shows a model with an "Unmatched Parenthesis". Inserting a closing parenthesis will restructure the expression tree according to operator precedence rules and remove the unmatched parenthesis element.

```
// "int1" -> incomplete
int1

// "int16" -> Variable declartion for 16 bit int without name
int16 <no name>;

// "int16 myfunc" -> Variable declaration with name
int16 myfunc;

// "int16 myfunc(" -> Function
int16 myfunc() {
} myfunc (function)
```

Figure 33. Inserting a C function in mbeddr restructures the AST as you type.

Figure 33 shows how a C function goes through various restructurings as you type.

4.1.4 Language demarcation: show how a combination of multiple languages in one program are disambiguated

MPS does not parse text and try to reconstruct a structure. Instead, every tree node creation binds a type to a node. In case of ambiguity the user can choose which type to create. For this reason, no disambiguation is needed for the type of a node and no special demarcation markers are needed.

Similarly, reference disambiguation is ensured by default since every node has a unique ID. Although the presentation of the reference can show a human readable name/identifier, the reference refers to the node with the given ID. A side effect of this is that removing an element named "A" and creating a new element named "A" will count as a new node and existing references will be broken. If desired, these references can also be automatically be re-bound to the new node ID.

```
// To demonstrate
int16 some_name() {
    return 42;
} some_name (function)

statemachine state_machine initial = initial {
    in event some_name() <no binding>

    var int16 some_name = 0

    state initial {
        on some_name [ ] -> some_name
    } state initial

    state some_name {
        do {
            some_name++;
            some_name();

            // Try to add some_name here, you will be
            // asked to disambiguate between the function
            // and the variable
        }
    } state some_name
}
```

Figure 34. A C module, containing a state machine, containing c statements demonstrating language boundaries. The name *some_name* is heavily overloaded to demonstrate disambiguation over language boundaries.

Figure 34 shows how various languages can be nested (c module, containing a state machine, containing c statements) and how references are resolved over these language boundaries. This example uses the same name ("some_name") for a state, an integer, a function, and a statemachine event. But since reference insertion binds to specific ids, references over language boundaries are disambiguated by default. The screenshot does not show this, but ctrl+clicking on the refer-

ences shows that all references point to the correct declaration.

4.1.5 Delayed decisions: show when the syntactic category of an expression is determined

At node insertion a defined concept must be inserted. However, it is possible to restructure the concepts as you type. Figure 33 shows how the syntactic category changes as you type, effectively delaying the final decision and providing a text-like experience.

If a user thinks in terms of the AST instead of linear text, it is also possible to immediately insert the desired concept by using code completion or typing the alias of the desired concept. In text terms this feels similar to using a template instead of typing sequentially.

4.1.6 End-user defined formatting: show if and how user can change the visual appearance of the program

In MPS, end-users can be given some control over the formatting. In the visual projection, for instance, end-users can move the nodes and the edges freely. There are some default layouts for the visual projection, but these layouts can be changed by the end-users.

Another example of end-user defined formatting is the insertion of empty lines in places where a statement is expected in mbeddr code. That means that the end-user can insert any number of empty lines in the body of a function, for instance. Subsequently, the empty line can be substituted by any sub-concept of *Statement*. To achieve this behavior the *Statement* concept has been declared with an editor that shows an empty line. Moreover, the *StatementList* concept contains an arbitrary number of *Statement* children, and by default, in MPS, at the press of an enter in a children collection (the statements in *StatementList*), a child concept is created (*Statement*). This results in showing an empty line in the viewer because the *Statement* concept is represented by an empty line. Moreover, by default, in MPS, a concept can be substituted by any of its sub-concepts, and hence, the *Statement* concept can be replaced by any of its sub-concepts.

Thus, the amount of freedom that the end-user has in defining its own formatting is decided, to some extent, by the language designer.

4.1.7 Specification of default formatting: support for pretty printing

JetBrains MPS has a default formatting for languages. The default formatting is simply following the structure of the abstract syntax of the language, printing the names of the instantiated concepts and their values, and also visually showing the parent-children relationship among concepts. An example of an expression with a defined formatting and a default formatting is shown in Figure 36.

```
void some_function() {
    int16 a;
    int16 b;

    // The defined editor for 2*3
    2 * 3;

    // The reflective (default) editor for 2*3. Right click the expression
    // and select "Show reflective editor"
    multi expression {
        left :
        2
        right :
        3
    };
} some function (function)
```

Figure 35. Defined and default formatting for a binary expression

A multitude of projections (formatting) can be defined for a given language and its concepts. A projection is defined in an editor component, where editor cells specify the location and other attributes of the concepts in the viewer. For instance, the editor component of the binary expression says that the left expression (with conditional parentheses around it) is followed by a constant (the symbol for the expression) and the right expression (again, with a conditional parentheses around it).

```
projection: [-? ( ( wrap FB% left % ? ) ) substitute grammar.constant ? ( ( wrap FB% right % ? ) ) -]
```

Figure 36. Defined editor for binary expression

Switching among projections does not modify the model itself, but only the representation of the model on the screen.

4.1.8 Formatting preservation: how is formatting preserved when the code is automatically restructured


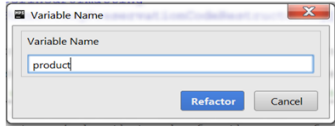
An example of a common refactoring is the introduction of a local variable in C functions [7]. The refactoring is triggered by selecting an expression in the body of a function, and pressing *CTRL-Alt-V*. As a result of this action, a window is prompted where the user introduces the name of the variable. The refactoring extracts the expression, places it in a local variable declaration and replaces all its occurrences in the body of the function with a reference to the variable. The process can be seen in Figure 37.

```

boolean isAtLimit(int8 v, int8 lim) {
    int8 val = measure(v * FACTOR);
    val = calibrate(val, v * FACTOR);

    return val >= lim;
} isAtLimit (function)

```

```

boolean isAtLimit(int8 v, int8 lim) {
    int8 product = v * FACTOR;
    int8 val = measure(product);
    val = calibrate(val, product);

    return val >= lim;
} isAtLimit (function)

```

Figure 37. Refactoring for introduction of a local variable

The refactoring for introducing a local variable is applicable to expression concepts that have a statement as an ancestor (see Figure 38). The body of the refactoring is straightforward: a new local variable declaration is introduced before the statement where the expression was selected and all the occurrences of the expression are looked up and replaced. Changing an expression with a variable declaration does not change the formatting because the change implies replacing a node in the tree with another, and the editor component will render the new node at the specific cell, without affecting other cells. The most intrusive operation is the introduction of the new local variable declaration in the existing statements, but this is the purpose of the refactoring itself.

```

refactoring introduceLocalVariable ( Introduce Local Variable ) overrides <nothing>
target: node<Expression>
    allow multiple: false
    isApplicableToNode(node)->boolean {
        node.ancestor<concept = Statement>.isNotNull;
    }

```

Figure 38. Implementation header of refactoring for introduction of a local variable

Other types of code restructuring are migration scripts in mbeddr. The situation is similar to the refactorings, the formatting of the existing code being unchanged (unless the migration itself changes the formatting).

5. Conclusions

The JetBrains Meta Programming System has significantly evolved during the years. Nowadays it is a powerful and flexible tool that can be used to address most of the Language Engineering challenges that have been brought forward in the LWC 2016.

6. Acknowledgments

Our thanks go out to Markus Völter and Kolja Dummann from the mbeddr team for providing us with good examples

from the mbeddr project. Without these, it would have taken us a lot more work to find nice examples in mbeddr or to construct such examples ourselves.

Moreover, we would like to thank Markus Völter for reviewing the paper and providing us with many helpful hints on how to improve it.

Finally, we would like to thank Remi Bosman from Sioux Embedded Systems for his contribution to the initial design proposal of this solution.

References

- [1] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.
- [2] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, Oct. 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167511. URL <http://doi.acm.org/10.1145/1167515.1167511>.
- [3] R. Solmi. Whole platform, 2005.
- [4] J.-P. Tolvanen. Metaedit+: Integrated modeling and metamodelling environment for domain-specific languages. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 690–691, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176676. URL <http://doi.acm.org/10.1145/1176617.1176676>.
- [5] F. Tomassetti, A. Vetrò, M. Torchiano, M. Voelter, and B. Kolb. A model-based approach to language integration. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, MiSE '13, pages 76–81, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-6447-8. URL <http://dl.acm.org/citation.cfm?id=2662737.2662755>.
- [6] M. Voelter. Integrating prose as first-class citizens with models and code. In *MPM@ MODELS*, pages 17–26, 2013.
- [7] M. Voelter. *Generic tools, specific languages*. TU Delft, Delft University of Technology, 2014.
- [8] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.
- [9] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. *Towards User-Friendly Projectional Editors*, pages 41–61. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11245-9. doi: 10.1007/978-3-319-11245-9_3. URL http://dx.doi.org/10.1007/978-3-319-11245-9_3.
- [10] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.
- [11] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. *Using C language extensions for developing embedded software: a case study*, volume 50. ACM, 2015.